



Calhoun: The NPS Institutional Archive
DSpace Repository

Theses and Dissertations

1. Thesis and Dissertation Collection, all items

2014

Development of a web-based distributed
interactive simulation (DIS) environment
using javascript

Hsiao, Chen-Fu

Monterey, California: Naval Postgraduate School

<http://hdl.handle.net/10945/43928>

This publication is a work of the U.S. Government as defined in Title 17, United States Code, Section 101. Copyright protection is not available for this work in the United States.

Downloaded from NPS Archive: Calhoun



Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

<http://www.nps.edu/library>



NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

**DEVELOPMENT OF A WEB-BASED DISTRIBUTED
INTERACTIVE SIMULATION (DIS) ENVIRONMENT
USING JAVASCRIPT**

by

Chen-Fu Hsiao

September 2014

Thesis Advisor:
Thesis Co-Advisor:

Christian J. Darken
Donald McGregor

This thesis was performed at the MOVES Institute

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE September 2014	3. REPORT TYPE AND DATES COVERED Master's Thesis	
4. TITLE AND SUBTITLE DEVELOPMENT OF A WEB-BASED DISTRIBUTED INTERACTIVE SIMULATION (DIS) ENVIRONMENT USING JAVASCRIPT			5. FUNDING NUMBERS	
6. AUTHOR(S) Chen-Fu Hsiao				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			10. SPONSORING/MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number ____N/A____.				
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			12b. DISTRIBUTION CODE A	
13. ABSTRACT (maximum 200 words) <p>This thesis investigated the current infrastructure for web-based simulations using the DIS network protocol. The main technologies studied were WebSockets, WebRTC and WebGL. This thesis sought readily available means to establish networks for interchanging DIS message (PDUs), so the WebSocket gateway server from Open-DIS project was used to construct a Client-Server structure and PeerJS API was used to construct a peer-to-peer structure. WebGL was used to create a scene and render 3D models in browsers. A first-person-shooter tank game was used as a test application with both WebSocket and WebRTC infrastructures.</p> <p>Experiments in this thesis included measuring the rate of sending and receiving DIS packets and analysis of the tank game by profiling tools. All the experiments were run on Chrome and Firefox browsers in a closed network.</p> <p>The experimental results showed that both WebSocket and WebRTC infrastructures were competent enough to support web-based DIS simulation. The results also found the significant differences of performance between Chrome and Firefox. Currently, the best performance is provided by the combination of Firefox using the WebRTC framework. The analysis of the tank game showed that most of the browser's computational resources were spent on the WebGL graphics, with only a small percentage of the resources expended on exchanging DIS packets.</p>				
14. SUBJECT TERMS Distributed interactive simulation (DIS), WebSocket, WebRTC, WebGL, client-server, peer-to-peer, web-based simulation			15. NUMBER OF PAGES 113	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UU	

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

**DEVELOPMENT OF A WEB-BASED DISTRIBUTED INTERACTIVE
SIMULATION (DIS) ENVIRONMENT USING JAVASCRIPT**

Chen-Fu Hsiao
Captain, Republic of China (Taiwan) Air Force
B.S., Chung Cheng Institute of Technology, 2007

Submitted in partial fulfillment of the
requirements for the degree of

**MASTER OF SCIENCE IN
MODELING, VIRTUAL ENVIRONMENTS, AND SIMULATION**

from the

**NAVAL POSTGRADUATE SCHOOL
September 2014**

Author: Chen-Fu Hsiao

Approved by: Christian J. Darken
Thesis Advisor

Donald McGregor
Thesis Co-Advisor

Christian J. Darken
Chair, MOVES Academic Committee

Peter J. Denning, Ph.D.
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This thesis investigated the current infrastructure for web-based simulations using the DIS network protocol. The main technologies studied were WebSockets, WebRTC and WebGL. This thesis sought readily available means to establish networks for interchanging DIS message (PDUs), so the WebSocket gateway server from Open-DIS project was used to construct a Client-Server structure and PeerJS API was used to construct a peer-to-peer structure. WebGL was used to create a scene and render 3D models in browsers. A first-person-shooter tank game was used as a test application with both WebSocket and WebRTC infrastructures.

Experiments in this thesis included measuring the rate of sending and receiving DIS packets and analysis of the tank game by profiling tools. All the experiments were run on Chrome and Firefox browsers in a closed network.

The experimental results showed that both WebSocket and WebRTC infrastructures were competent enough to support web-based DIS simulation. The results also found the significant differences of performance between Chrome and Firefox. Currently, the best performance is provided by the combination of Firefox using the WebRTC framework. The analysis of the tank game showed that most of the browser's computational resources were spent on the WebGL graphics, with only a small percentage of the resources expended on exchanging DIS packets.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I.	INTRODUCTION.....	1
II.	BACKGROUND.....	5
A.	MOTIVATION.....	5
B.	TECHNOLOGIES.....	6
1.	Networking	6
2.	DIS Protocol	7
3.	JavaScript.....	8
4.	Web Server with WebSockets.....	9
5.	WebRTC.....	9
6.	WebGL	12
7.	JSON DIS Format.....	12
8.	Binary DIS Format	13
III.	EXPERIMENTAL DESIGN	15
A.	NETWORKING FRAMEWORK	15
1.	Client-Server Architecture (WebSocket).....	15
a.	<i>Open-DIS WebSocket Gateway Server.....</i>	15
b.	<i>Framework.....</i>	16
2.	Peer-to-Peer Architecture (WebRTC)	16
a.	<i>PeerJS.....</i>	17
b.	<i>Framework.....</i>	17
B.	PERFORMANCE WITH WEBGL.....	18
1.	Without WebGL Elements	18
2.	With WebGL Elements	19
a.	<i>3D Models</i>	19
b.	<i>Game Design</i>	19
C.	PERFORMANCE IN DIFFERENT BROWSERS.....	21
IV.	IMPLEMENTATION	23
A.	SERVER PREPARATIONS	24
1.	WebSocket Gateway Server and Web Server.....	24
2.	WebRTC: PeerServer and Primary Peer	25
B.	BROWSER INITIATIVE PROCESSES AND BEHAVIORS	25
1.	Import Libraries	26
a.	<i>WebSocket.....</i>	26
b.	<i>WebRTC.....</i>	26
c.	<i>WebGL</i>	26
d.	<i>Others</i>	26
2.	Checking Web Browser Brand and Initiating WebSocket or WebRTC	27
a.	<i>WebSocket.....</i>	27
b.	<i>WebRTC.....</i>	28
3.	Create Canvas, Scene, Camera, and Own Entities	30

4.	Game Control and Graphics Rendering.....	31
5.	Convert Coordinate Systems.....	33
6.	Send DIS Packets	35
7.	Receive and Decode DIS Messages.....	37
a.	Receive by WebSocket.....	37
b.	Receive by PeerJS	37
c.	Decoding DIS Messages.....	38
8.	Miscellaneous	40
a.	Loading 3D Objects	40
b.	Entity Collision Detection.....	41
c.	Checking Existence	42
d.	Dead Reckoning.....	42
V.	PERFORMANCE TESTS.....	43
A.	SENDING AND RECEIVING ABILITY	43
B.	PROFILING JAVASCRIPT PERFORMANCES.....	51
C.	NETWORK LOADING TIME.....	55
VI.	RESULT DISCUSSIONS	59
A.	SENDING AND RECEIVING EFFICIENCY	59
B.	JAVASCRIPT PERFORMANCE.....	62
1.	Chrome Developer Tools	62
2.	Firefox Developer Tools	64
C.	LOAD OF NECESSARY ARCHIVES.....	65
D.	GAME SCALABILITY	66
VII.	CONCLUSION AND FUTURE WORKS	69
A.	CONCLUSION	69
B.	FUTURE WORKS	71
1.	Performance Tests and Web Applications on Mobile Devices	71
2.	Improvement of WebRTC Sending Capability in Chrome Browser and Benchmark Test	72
3.	Disadvantages of Web-based Simulation.....	73
4.	Feasibility of Other Web-based DIS Applications.....	73
5.	Performance Tests in Public Networking Environments ...	73
	APPENDIX A. TABLE OF ENTITY STATE PDU FIELDS	75
	APPENDIX B. FIREFOX DEVELOPER TOOLS PERFORMANCE PROFILE RESULTS	77
A.	APPENDIX B-1. FIREFOX PROFILING OF SENDING ESPDUS.....	77
B.	APPENDIX B-2. SENDING SECTION IN APPENDIX B-1.....	78
C.	APPENDIX B-3. FIREFOX PROFILING OF RECEIVING ESPDUS ..	79
	APPENDIX C. TANK GAME RESULTS.....	81
A.	APPENDIX C-1. FIREFOX PROFILING IN THE WEBSOCKET FRAMEWORK	81

B. APPENDIX C-2. FIREFOX PROFILING IN THE WEBRTC FRAMEWORK	82
APPENDIX D. NETWORK LOADING TIME	83
APPENDIX E. COMPARISONS OF RECEIVING FIRST TEN 10,000 DIS PDUS IN THE LINEAR FORM.....	85
APPENDIX F. SCREENSHOTS OF MULTIPLAYERS IN THE TANK GAME.....	87
LIST OF REFERENCES.....	89
INITIAL DISTRIBUTION LIST	93

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

Figure 1.	JSEP architecture(from Dutton, 2013).....	11
Figure 2.	WebSocket networking framework.	16
Figure 3.	WebRTC networking framework.....	18
Figure 4.	Tank game design.	21
Figure 5.	Experimental environment.	24
Figure 6.	Install and execute PeerServer.	25
Figure 7.	Relationships of importing libraries and other JavaScript programs. ...	27
Figure 8.	Creating WebSocket code.....	27
Figure 9.	Initiate a PeerJS client.....	28
Figure 10.	Architecture and sequences of creating peer connections.	29
Figure 11.	Creating canvas in a web page.	30
Figure 12.	Creating scenes and adding graphical elements.....	31
Figure 13.	Creating tank meshes and ESPDUs in ourEntity objects.	31
Figure 14.	Function of keydown events.	32
Figure 15.	Function of keyup events and window event handlers.	33
Figure 16.	Earth centered, earth fixed; and east, north, up coordinates ("Axes conventions," n.d.).....	34
Figure 17.	Conversion between ECEF and ENU.....	35
Figure 18.	Heartbeat function.	36
Figure 19.	PeerJS sending method.	36
Figure 20.	Set format and attach functions.....	37
Figure 21.	PeerJS event listener.	37
Figure 22.	Example code of receiving events function.	38
Figure 23.	Example code of dis.PduFactory().	39
Figure 24.	Code of updating entity states and 3D model.....	41
Figure 25.	MeasureDIS function for sending DIS packets.	44
Figure 26.	Measuring function of receiving DIS packets in WebSocket.....	44
Figure 27.	Measuring function of receiving DIS packets in PeerJS.	45
Figure 28.	Purely sending and receiving capabilities.....	46
Figure 29.	Purely receiving capabilities.	47
Figure 30.	Comparisons of receiving capabilities between the presences of WebGL.	48
Figure 31.	Running measureDIS function for a long period-of-time.....	52
Figure 32.	Running measureDIS function for a short period-of-time.....	52
Figure 33.	Tank game profile using WebSocket framework in Chrome.	54
Figure 34.	Tank game profile using WebRTC framework in Chrome.	55
Figure 35.	Screenshot of tank model.....	56
Figure 36.	Screenshot of terrain model.	57
Figure 37.	Comparisons of receiving first ten 10,000 DIS PDUs.	61
Figure 38.	A Chrome profiling sample of invoking one renderFunc function.	63
Figure 39.	A Chrome profiling sample of invoking one websocket.onmessage and one heartbeat function.....	64

Figure 40.	A Firefox profiling sample of invoking renderFunc functions.	65
Figure 41.	Firefox profiling of sending ESPDU in the WebSocket framework.	77
Figure 42.	Sending section in Appendix B1.....	78
Figure 43.	Firefox profiling of receiving ESPDU in the WebSocket framework....	79
Figure 44.	Firefox profiling of the tank game in the WebSocket framework.....	81
Figure 45.	Firefox profiling of the tank game in the WebRTC framework.	82
Figure 46.	Network loading time.....	83
Figure 47.	The comparisons of receiving first ten 10,000 DIS PDUs in the linear form.	85
Figure 48.	Screenshots of multiplayer in the tank game I.....	87
Figure 49.	Screenshots of multiplayer in the tank game II.....	87
Figure 50.	Screenshots of multiplayer in the tank game III.....	88
Figure 51.	Screenshots of multiplayer in the tank game IV.....	88

LIST OF TABLES

Table 1.	TCP, UDP, and SCTP comparison (from Ristic, 2014).....	11
Table 2.	t-tests of receiving capability in WebSocket and WebRTC frameworks without WebGL components. The experiment at left had the higher performance.	49
Table 3.	t-tests of receiving capability between browsers without WebGL components in WebSocket framework.	49
Table 4.	t-tests of receiving capability between browsers without WebGL components in WebRTC framework.....	50
Table 5.	t-test between presences of WebGL components.....	51
Table 6.	Fields of entity state PDU.	76

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

API	Application programming interface
CC	Chrome sent to Chrome browser
CF	Chrome sent to Firefox browser
CPU	Central processing unit
CSS	Cascading style sheets
DIS	Distributed interactive simulation
DTLS	Datagram transport layer security
EBV	Enumeration and bit encode values
ECEF	Earth-centered, Earth-fixed
ENU	East, north and up
ESPDU	Entity state PDU
FC	Firefox sent to Chrome browser
FF	Firefox sent to Firefox browser
FSP	Frames per second
GPU	Graphics processing unit
GUI	Graphical user interface
HOV	High-occupancy vehicle
HTML	Hypertext markup language
IDE	Integrated development environment
IEEE	Institute of electrical and electronics engineers
IETF	Internet engineering task force
JBUS	Joint simulation bus
JCATS	Joint conflict and tactical simulation
JIT	Just-in-time
JSEP	JavaScript session establishment protocol
JSON	JavaScript object notation
LVC	Live, virtual, and constructive
MOVES	Modeling, virtual environments and simulation
OneSAF	One semi-automated forces
OS	Operating system

PDU	Protocol data unit
SCTP	Stream control transmission protocol
SDP	Session description protocol
SISO	Simulation interoperability standards organization
SSL	Secure sockets layer
SVG	Scalable vector graphics
TCP	Transmission control protocol
TCP/IP	Transmission control protocol/Internet protocol
UDP	User datagram protocol
VBS2	Virtual battlespace 2
W3C	World wide web consortium
WebGL	Web graphics library
WebRTC	Web real-time communication
WWW	World wide web

ACKNOWLEDGMENTS

I would like to express my sincere appreciation and thanks to my Advisor, Professor Christian Darken, and co-Advisor, Donald McGregor, for their expert guidance and endless patience. Their assistance on web-based 3D graphics knowledge, DIS networking interoperability and other technique support was priceless. In addition, I also want to thank the MOVES faculty who prepared an admirable laboratory for supporting my thesis experiments. Finally, a special thanks to my family, JETC colleagues, and friends, who gave me invaluable support and help, so I could focus on achieving my study and research.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

In the military domain, simulation is widely used for training, education, and analysis. Many simulation systems have been developed to support different missions, such as flight simulators for training and educating pilots, or Virtual Battlespace 2 (VBS2) for educating platoon leaders. After multiple simulation systems have been established, users may want to integrate them for joint training purposes. Therefore, the requirement for interoperability among systems becomes an important topic.

In the 1990s, the Simulation Interoperability Standard Organization (SISO) reached an agreement on a distributed interactive simulation (DIS) protocol. SISO also had the DIS protocol ratified as an IEEE standard, so it is now available for anyone to read and implement. Currently, the DIS protocol has been used to develop many simulation systems or communicate among existing systems by formatting the exchanged data. There are many existing simulation systems that have implemented the DIS protocol by different programming language (Rogerson, 1997; McGregor, Brutzman & Grant, 2008). One purpose of military simulation is to construct a live, virtual and constructive (LVC) environment for military training, education, and analysis (Farlane et al., 2004). “*Live*” refer to real people operating real systems such as real pilots flying F-16 fighters. “*Virtual*” refers to real people operating simulated systems, such as real drivers in high-occupancy vehicle (HOV) simulators. “*Constructive*” describes a simulation in which both people and systems are simulated such, as a simulated opponent force—an opponent fighter with AI—in a simulated system. In order to achieve LVC architecture, simulation systems have to exchange information with each other, and DIS protocol represents a standardized format for communicating data.

With the developments of web technology and the improvements of computer performance, more and more applications can be run in web browsers. This thesis implemented several web technologies that mainly included

WebSocket, WebRTC, and WebGL to discuss the feasibility of applying DIS protocol to multiple-user web-based simulation. The client/server and peer-to-peer architectures can be used to develop web-based simulations, and their respective technologies are WebSocket and WebRTC. This thesis compared the performance of sending and receiving DIS messages in these two different networking architectures, and incorporated WebGL components to develop a first-person-shooter game to analyze the performance in different browsers.

This thesis also discussed the advantages of web-based simulation. The features of web-based simulation are easier to upgrade (centralized content), are cross-platform, and are widely accessible via computers, tablets, and mobile devices. For example, when people want to execute simulation systems, they have to set up an environment for execution. This environment includes a physical desktop setup, an operating system install, a simulation software install, and a peripheral device setup. Typically, computers have pre-installed operating systems with peripheral device setups. The simulation system, however, has to be installed additionally, and each one has its own compatible operating system (e.g., Windows 7, Windows Server 2012, UNIX, and different versions of Linux). When users want to run a specific simulation system on computers, they have to check and configure all operating systems and system setups before running the simulation. The computer preparation for running this specific simulation system may be a tedious and inefficient process.

Web-based simulation can relieve the above situation and increase the efficiency of system readiness. Nowadays, the major browsers that most people use on desktop and laptop computer are Chrome, Firefox, Internet Explorer, and Safari. All the experiments, comparisons and analysis in this thesis, however, were run in Chrome and Firefox because both Chrome and Firefox support the same web technologies. In addition, both Chrome and Firefox provide installers for mainstream operating systems such as Windows, Mac OS, and Linux, so web-based simulation can run on almost every operating system through these two platforms (Chrome and Firefox).

The thesis begins by describing the motivation and the benefits of browser-based simulation and outlining the technologies for constructing the infrastructures. Next, the thesis describes design and implementation for testing the performance among several variables. The latencies and the results were discussed at the end with commentary for future work.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BACKGROUND

In the past, developing a networked and interactive simulation system usually required complex programming skill. Many programming language and technologies have been developed to reduce the complexity of programming. The following sections describe motivations and several present technologies for developing a small-scale and interactive simulation.

A. MOTIVATION

With the prosperity of the Internet, more and more web applications have been developed. With the development of web technologies and the increasing of computational power, the possibility of developing interactive and complicated applications in web is promising, and because of the elements of interaction and logical computation capability, web-based simulation systems are more achievable. Web-based simulation is exploiting resources and technologies offered by the World Wide Web (WWW) to represent traditional simulation systems (Fishwick, 1996). In other words, web-based simulation uses web browsers as graphical interfaces to link users and simulation systems. The benefits of exploiting web-based simulation are cross-platform, collaboration, model reusing, deployment, wide availability, versioning control, etc. Modern browser companies provide browser installers for different operating systems, and most modern browsers support the same web technologies (e.g., JavaScript, JSON, binary data format, WebGL, and WebSocket), which increases the cross-platform capability. Collaboration is useful in that web simulations can be designed to help people involved in a common task to accomplish goals. Model reusing is that many existing 3D graphic formats such as .dae, .obj, .blend can be imported into web applications, so software engineers can focus on the design of web applications rather than making 3D models. Web applications also increase their accessibility and deployment to users because every operating system has its compatible browsers that support the same web technologies. On

server side, web applications can ease the burden of versioning control for system managers. They only need to update archives on server and ask users to reload and refresh web contents. Additionally, web applications are easy to execute because browsers do not need further configuration or plug-in to run applications. For the military domain, state-of-the-art web technologies can not only construct web-based simulations (McGregor & Brutzman, n.d.), but also enable communication between web applications such as Google Maps and existing simulation systems such as VBS2, OneSAF, and JCATS via proper gateways (McGregor, Blais, & Brutzman, n.d.). The ability of interoperating with other systems increases the practicability of web-based simulation. Furthermore, inventors of web technologies also ease the learning curve of developing web applications. HTML5 and JavaScript are easy to learn and practice programming languages, and developers do not require compilation JavaScript manually or extra procedures for users to run web applications.

B. TECHNOLOGIES

In order to establish a web-based simulation with the abilities of easy deployment and maintenance, many elements have to be applied. These include networking architecture, DIS protocol, JavaScript, web server with WebSocket, Web Real-Time Communication (WebRTC), WebGL, JSON format DIS, and binary format DIS messages.

1. Networking

Networking is separated into five layers: application, transport, network, link, and physical. Most of the applications are implemented in the application layer over TCP/IP protocol, a framework for communication between devices. TCP/IP is a software concept that allows one machine to send bytes to another, without knowing the content or meaning of those sent bytes. Applications are used to interpret TCP/IP sending data. Modeling and simulation (M&S) networking protocols are standardized in the application layer, and all the M&S applications such as JCATS and OneSAF relay on this layer, too.

Network architecture usually contains more than two hosts (Steed & Oliceira, 2010, Ch. 4) in military simulations, so it is important to define models for multi-host connections: simple peer-to-peer, peer-to-peer with master host, peer-to-peer with rendezvous server, and client/server. Simple peer-to-peer communication means that each host has a configuration file to record necessary information of all the participants (as well as address/port data) and then sends updates to all the other hosts within the network group. Peer-to-peer with master means that one of the participants will be the rendezvous access point. The rendezvous point has a well-known IP and port number, so any host who wants to join the network can obtain other participant information from the master. The advantage over the simple peer-to-peer model is that there is no need to collect every participant's IP address, port number and other information beforehand. The peer-to-peer with rendezvous server model uses a server that has information for all participants. Every host who wants to join the network has to connect to the rendezvous server first to ask for network environment information. The difference between peer-to-peer with master and peer-to-peer with rendezvous server is that the master is one of the network participants but the rendezvous server, who is not a participant in the network, is only the distributor with all the hosts' information for a new host who wants to join the network. Client-server architecture means that each host connects to the server, and the server is responsible for every communication between hosts.

2. DIS Protocol

DIS is a standardized protocol used to communicate and exchange information in a multiplayer simulation system (*DIS IEEE Std 1278.1-2012—Standard for Distributed Interactive Simulation—Application Protocols*, 2012). It also allows two or more different types of simulators to interact or interoperate, especially those simulators that have human-in-the-loop elements. For example, a joint forces exercise uses ship simulators and flight simulators to train the capability of cooperation. The method that DIS uses to interchange information is based on protocol data unit (PDU) messages; there are many kinds of PDUs,

such as entity state PDU (ESPDU), detonation PDU and entity damage status PDU. Different PDUs have different lengths and fields to contain the variety of information, so simulation system can communicate with each other by sending and receiving PDUs. Although there are many kinds of PDUs with different fields for restoring information, every PDU starts with the same header that is used to differentiate PDU types when receiving DIS packets. Appendix A displays a table containing entity state PDU fields, and shows that the common PDU header used the first 96 bits. The table also shows the other fields of ESPDU. ESPDU is the most frequent PDU in DIS simulation, and it is used for interchanging the information of entity's state. The information includes entity ID, entity type, location and orientation in the simulated world, entity appearance, entity capabilities, etc.

The DIS uses a heartbeat strategy, in which entities periodically send out their PDUs even if they do not change their properties or states. A critical issue, however, is the rate at which each entity sends its PDUs—sending data too often would cause networking congestion, latency or losing data, etc. In a large-scale scenario, some of the entities are relatively slow or even in a static state (e.g., tanks or infantry), but some of the entities move quickly (e.g., airplanes). Therefore, the developer has to set different update rates for different kinds of entities to avoid redundant DIS packets transferring in network. Another principle is not to let the late-joining hosts wait too long for those slow-moving entities. DIS simulation is mainly used in military domains, and it usually assumes that DIS simulation is implemented in a high-performance intranet with high security, one in which participants will not send fake or swindling messages. Additionally, in order to simplify data transfer, DIS usually broadcasts or multicasts its PDUs based on UDP socket.

3. JavaScript

JavaScript is the scripting language of the web, and all modern web browsers support JavaScript (w3schools, n.d.). JavaScript lets browsers have the

capability of logical computation and client-side scripting that let users change or interact with web content depending on user input, which is in contrast with server-side scripts such as PHP, Java and Python (Flanagan, 2011; Powers, 2010).

4. Web Server with WebSockets

Traditional web servers only talks to a user's page once, when user loads the content from the web server; this is because the web pages were designed to be static originally. This causes a problem for web pages that need highly interaction with users. With WebSocket, JavaScript opens a TCP socket to establish a communication channel to the server and request this special type of TCP socket to be opened for delivering arbitrary application protocols between client and server (Grigorik, 2013, Ch. 17). This specialized TCP socket can remain open, so web pages can keep updating its contents periodically with some JavaScript program. Furthermore, WebSocket also has features of low latency compared to HTTP polling; and higher bandwidth. WebSocket makes possible to run interactive and real-time applications in web browsers.

5. WebRTC

WebRTC—whose API is defined by W3C and protocol is defined by IETF—is a plugin-free real-time communication API that is used for high-quality audio, video and data communication with low cost (“WebRTC,” n.d.). The features of WebRTC are binding a UDP socket, peer-to-peer connection, and cross-platforms interaction. Four main tasks for WebRTC are acquiring audio and video, establishing a connection between peers, communication audio and video, and communicating arbitrary data. In order to achieve the above tasks, three main JavaScript APIs—MediaStreams (a.k.a. getUserMedia), RTCPeerConnection and RTCDataChannel are applied. So far, WebRTC can run in Chrome, Firefox, and Opera.

MediaStreams, acquiring audio and video, represents a stream of synchronized media, and it can contain multiple audio and video tracks. To

obtain a `MediaStream`, JavaScript provides a method called `navigator.getUserMedia()`. When invoking `navigator.getUserMedia()`, a web browser will pop out a HTTPS prompt and ask the user's permission for accessing the camera and microphone. While developing web apps using WebRTC API, developers can combine video and audio streams with JavaScript Canvas and WebGL. In the mobile devices with multiple cameras or microphones, users can choose input devices through WebRTC API. Another function of WebRTC is `getUserMedia()` for capturing user's screen, which is useful for desktop sharing or online remote teaching.

`RTCPeerConnection` is implicitly used for audio and video communication between peers. A web browser takes the media streams from `getUserMedia()`, plugs them into the peer connection and sends them off to the other side. The peer connection is responsible for many things (e.g., signal processing, codec handling, peer to peer connection, security, and bandwidth management). WebRTC hides most of the complexity from web developers, so developers can get media streams easily and plug them into peer connection. `RTCPeerConnection`, however, needs servers to broker a connection when the peers want to make connections. Therefore, a process called signaling is applied, which is like making a telephone call. When a caller makes a phone call, the telephone network sends a message to a callee. After the callee answers the call, the callee sends a message back to activate a connection. WebRTC does a similar thing. When a peer wants to establish a connection, its application first signals to the server and then sends session description objects that contain parameters and Internet information to the browser for setting up the peer-to-peer route. Figure 1 describes the technique of making peer-to-peer connections between browsers. WebRTC allows users using any mechanism, protocol, or even JSON to make connections to maximize compatibility with established technologies, which is defined by JavaScript session establishment protocol (JSEP) (Dutton, 2013).

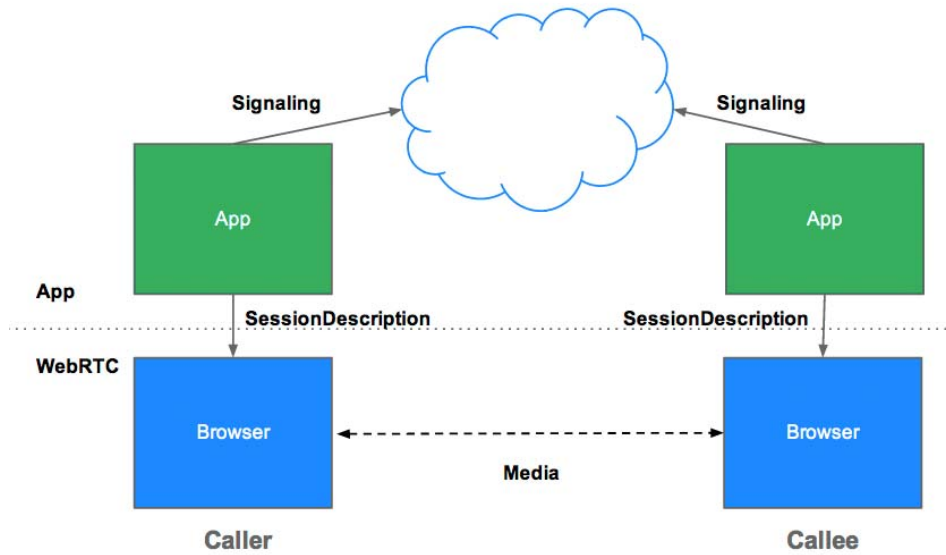


Figure 1. JSEP architecture(from Dutton, 2013).

RTCDatChannel is a bidirectional communication of arbitrary data between peers, and all the data is encrypted by Datagram Transport Layer Security (DTLS), which is a derivative of SSL. RTCDatChannel is much like a WebSocket, but it relies on the Stream Control Transmission Protocol (SCTP), which runs on top of the UDP socket. A comparison of TCP, UDP, and SCTP is in Table 1. Based on the features of SCTP, RTCDATACHannel is suitable for arbitrary data transfer or multiplayer gaming.

	TCP	UDP	SCTP
Reliability	reliable	unreliable	configurable
Delivery	ordered	unordered	configurable
Transmission	byte-oriented	message-oriented	message-oriented
Flow control	yes	no	yes
Congestion control	yes	no	yes

Table 1. TCP, UDP, and SCTP comparison (from Ristic, 2014).

6. WebGL

Web graphics library (WebGL) is a JavaScript API for rendering 2D or 3D graphics in any compatible browser (i.e., most modern browsers such as IE v11.0, Opera v23.0, Chrome v36.0, Firefox v31, or Safari v7) (Deveria, n.d.-a). WebGL can be used for processing 2D images, creating visually 3D graphics and visualizing different kinds of data. For example, D3.js is a JavaScript API that visualizes varieties of data in browsers by using HTML, SVG, and CSS. Many WebGL libraries are game engines for rendering graphics in browsers; these include pixi.js, Phaser, and three.js. WebGL provides the web pages with the capability of efficiently creating interactive 2D and 3D graphics to simulate objects in web-based simulations.

7. JSON DIS Format

JavaScript Object Notation (JSON) is a text-data format that facilitates data interchange between different languages (“JSON,” n.d.). The purpose of JSON is to store and exchange text information. It is like XML, but smaller, faster, and easier to parse. JSON is used to describe data objects not only for JavaScript, but also for other programming languages. It is language-independent because most of the programming languages have their own methods and libraries to parse JSON text. JSON and JavaScript syntactically use the same method to describe objects, so when JavaScript receives JSON format files, JavaScript uses a built-in `eval()` function to generate JavaScript objects. JSON format is based on two kinds of structure: pairs of name and value, and ordered lists. Pairs of name and value is similar to object, record, struct, dictionary, hash table, keyed list, or associative array in other languages. Ordered list is like array in other languages.

DIS protocol can be formatted in the form of JSON, which uses a tag-value approach to make JavaScript objects containing PDU information. This DIS JSON can be sent by WebSocket or WebRTC API after invoking `JSON.stringify()`, a JavaScript method, to convert a JavaScript object to JSON. Another browser

can receive this DIS JSON and decode as a JavaScript object. Although JSON exchanges data well, it has some inevitable drawbacks. JavaScript object uses a tag-value approach, so both publisher and subscriber must have agreement regarding field names in the messages. This feature increases the flexibility of creating objects, but it uses more bandwidth in messages.

8. Binary DIS Format

DIS protocol has been approved by IEEE, and it is widely used in many existing simulation applications. Using binary format to exchange messages between browsers has a better performance than JSON. Not only the message size in DIS binary is smaller than JSON with full DIS messages, but also the decoding speed in DIS binary is faster than JSON messages (McGregor et al., n.d.). Another advantage of binary DIS format is that DIS is a standardized protocol, and using it eliminates the intermediary gateways for protocol translation. Additionally, many existing gateways convert different protocols into DIS protocol such as JBUS—Joint Simulation Bus—and this increases the interoperability between existing simulation systems and web-based simulations.

Using the above elements—which included networking, DIS protocol, programming language, web server with specialized TCP socket, WebRTC, 3D graphics, and DIS JSON formatting or binary formatting message—it is possible to create an interactive web-based simulation with human-in-the-loop features.

THIS PAGE INTENTIONALLY LEFT BLANK

III. EXPERIMENTAL DESIGN

To understand the performance of running DIS in browsers, this thesis created an experimental design with several variables that included networking framework (WebSocket and WebRTC), application of WebGL and different browsers. The experimental devices included a wired router, two desktop computers as clients and one laptop computer as WebSocket server and WebRTC server. The specifications of the two desktop computers are as follow: Intel® Xeon® CPU E5-1603 0 @ 2.8 GHz, 6 GB memory, NVIDIA Quadro 4000 graphics card, and Windows 7 64-bit operating system. The specifications of the server are as follow: Intel® Core™2 Duo CPU L9400 @ 1.86 GHz, 2 GB memory, NVIDIA GeForce 320 M graphics card and Windows 7 32-bit operating system.

A. NETWORKING FRAMEWORK

There are many ways to make communication among computers, such as client-server architecture or peer-to-peer architecture. Because of the emergence of web technologies (WebSocket and WebRTC), many networking architectures can be applied to web-based applications.

1. Client-Server Architecture (WebSocket)

WebSocket is a TCP-based socket, which means it has all the features of TCP socket such as reliable and ordered data interchange, and it can be used to transport arbitrary data type for different web applications.

a. *Open-DIS WebSocket Gateway Server*

Open-DIS is an open-source implementation of the DIS protocol in many languages (McGregor, Grant, Smith, Harder & Snyder, n.d.). It was developed mainly by the Modeling, Virtual Environment, and Simulation (MOVES) Institute at the Naval Postgraduate School. From the Open-DIS official website, users can download a “javascript.zip” archive to obtain WebSocket gateway server. The archive has example applications including sending and receiving native DIS

messages and WebSocket/Javascript/WebGL applications. For this experiment, the WebSocket application was run in NetBeans, which is a free integrated development environment (IDE) for developing desktop, mobile and web applications with JAVA, C++, HTML5, JavaScript and more.

b. Framework

The DIS implementation of WebSocket gateway server is client-server architecture, shown in Figure 2. Every DIS message that is sent from the web browser must go through the WebSocket gateway server. The server will repeat received DIS messages to all connected web browsers. Because all the client browsers are connected to the server, the server plays a vital role in a client-server web-based simulation. If the server loses its networking connection, all the browsers will lose their ability to interoperate, and then the web-based simulation will become a single-player game or crash.

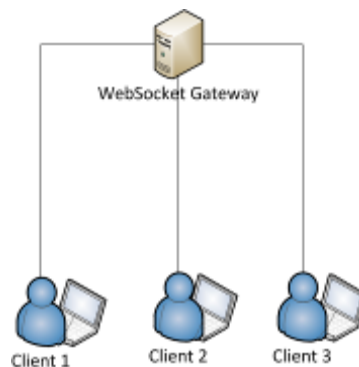


Figure 2. WebSocket networking framework.

2. Peer-to-Peer Architecture (WebRTC)

Unlike the WebSocket, WebRTC uses UDP sockets that have no flow control and no congestion control. They are also unreliable and offer unordered data exchange to transfer data. The WebRTC data channel, however, uses SCTP—which is a protocol on top of the UDP sockets—to configure data reliability and to order and control data flow and congestion. The networking architecture of WebRTC is like a peer-to-peer with rendezvous server model.

a. *PeerJS*

PeerJS is an application programming interface (API) based on WebRTC (Bu & Zhang, n.d.). It wraps WebRTC implementation and provides a fully documented and easily configurable API for developers to create peer-to-peer connections in web browsers with nothing but a unique ID. Client browser creates a unique ID and connects to PeerServer, and the server uses this ID to identify and deliver connection information to the prospective peers. The ID is formed with numbers and letters. If a client browser connects to a PeerServer without a unique ID, the PeerServer would create an ID for this active client. If a client-browser connects to a PeerServer with an ID that has been used for other client browser, this client would connect as failed, and the server would respond with error messages to the client.

On each connection between a pair of browsers, audio, video, and arbitrary data can be sent. Although WebRTC is peer-to-peer connections, the client's browser must first signal to a PeerJS server to get connection information that is based on a session description protocol (SDP). WebRTC uses SDP to describe a session profile, which contains information such as transportation address, media information, and related metadata. Browsers use this session profile to create peer connections. PeerJS provides PeerServer on Cloud, a public server that everyone can use by registration on the PeerJS Website, and PeerServer application for installation in a private network.

b. *Framework*

This thesis used PeerJS to create a data channel to interchange DIS messages between peers. The framework is in Figure 3. Before a peer-to-peer connection, each peer has to initially connect to a PeerServer to get another peer's information and a web server to receive web content. Once the peer browsers construct data channels and receive web content, the peers no longer need the PeerServer.

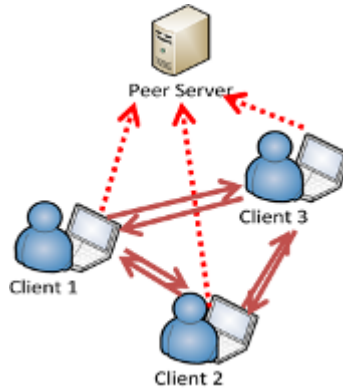


Figure 3. WebRTC networking framework.

B. PERFORMANCE WITH WEBGL

WebGL is one of the critical elements in web-based simulation. Users feel more immersed if a simulation has lifelike models, especially 3D models in a vivid virtual environment. Running 3D graphics, however, is resource consuming, which means computers have to devote lots of CPU or GPU resources to computing the change of 3D graphics (e.g., movement, scaling, rotation). Therefore, this thesis designed a comparison for applying WebGL or not. All the experimental 3D models were made with Blender, which is free and open-source software for computer 3D graphics. Blender has an add-on function to export 3D graphics, and a library implemented WebGL (three.js), which has loaders for importing .js models into web pages.

1. Without WebGL Elements

Purely DIS messages send and receive without WebGL elements in different networking frameworks. In order to find out the capability of sending and receiving speed in modern browsers, two browsers were opened on two computers with the same hardware devices; one was for sending DIS packets and the other was for receiving DIS packets. The sending browser repeatedly converted an ESPDU JavaScript object to binary format DIS messages before sending those messages. The receiving browser decoded the binary message,

converted it a JavaScript object and distinguished PDU types every time it received DIS messages.

2. With WebGL Elements

WebGL enables web browsers to render 3D graphics without plugins, but it also takes a lot of computational resources. This thesis created a demonstrating game that used WebGL and the three.js library to render 3D graphics and DIS PDUs to exchange entity information.

a. 3D Models

This thesis designed three major 3D models (terrain, tank and tank ammunition) to demonstrate the feasibility of web-based simulation. The demonstration was a simple and first-person-shooter tank game. The players controlled a single tank to search, aim, and shoot other tanks in a virtual environment in a web browser. This game was designed for multiple players. Each player, and each browser page, created its own virtual environment (including scene, skybox, light, camera, terrain, a controllable tank, tank ammunition, etc.) when the browser connected to a web server and got the game html file. Browsers started sending and receiving DIS messages once they connected to a WebSocket gateway server or established a WebRTC data channel. Other tank models and ammunition models were created when a browser received ESPDUs whose entity IDs were new to that browser, which meant that each opponent tank model had a corresponding entity ID. If an incoming ESPDU's entity ID already had a representative 3D object, the browser would update the state and location of this 3D object in the virtual world.

b. Game Design

There are three different types of DIS PDUs in this tank game: entity state PDU (ESPDU), collision PDU and fire PDU. Figure 4 describes the mechanism of this tank game. The upper section is about sending DIS PDUs, and the lower describes receiving DIS PDUs. When the game started and a controllable tank

was created in the scene, the tank could move and search for targets. At the same time, the tank issued ESPDUs every ten milliseconds, which was the same rate at which WebGL updated its canvas.

The ESPDUs are the major interchanging PDUs in DIS simulations, and contain all the basic information of entities (e.g., entity ID, entity type, entity location, and entity orientation). The collision PDU contains information about collision events, and is issued when a collision happens between two simulated entities or between a simulated entity and another object in the virtual world. In this tank game, a collision PDU was issued only when a tank's projectile hit another tank. The fire PDU is used to support visual, aural, and other effects, and identify an entity that fired a weapon or expendable. The fire PDU in this game, however, was only used to communicate firing events and showed the firing information on screen.

To play this game, the player has to eliminate all other tanks in the virtual battlefield. If a tank gets hit while searching for targets, it will issue a collision PDU. The information in a collision PDU includes the issuing entity ID, colliding entity ID, event ID, location, etc. If a tank finds a target and shoots it, the tank will issue a fire PDU regardless of whether it hits its target. A fire PDU contains the fields firing entity ID, target entity ID, location, etc. for describing the firing event. If a tank hits a target, the tank's projectile will issue a collision PDU. In the receiving section, when the browser receives a DIS message, the message will be decoded and categorized into ESPDU, collision PDU or fire PDU. If a browser receives an ESPDU and the ESPDU's entity ID is new to this browser, the browser loads a 3D object to represent this entity ID. Otherwise, the browser updates the state and location of this 3D object. If browsers receive fire PDUs, they display the fire information in the upper-left corner of the window. If a browser receives a collision PDU, it checks whether its controllable tank issued a collision PDU within two seconds (a game setting). If the answer is yes, the tank is killed, and the game is over. If no, there might be some latency or lag in

networking traffic that created an asynchronous situation, so the game would continue.

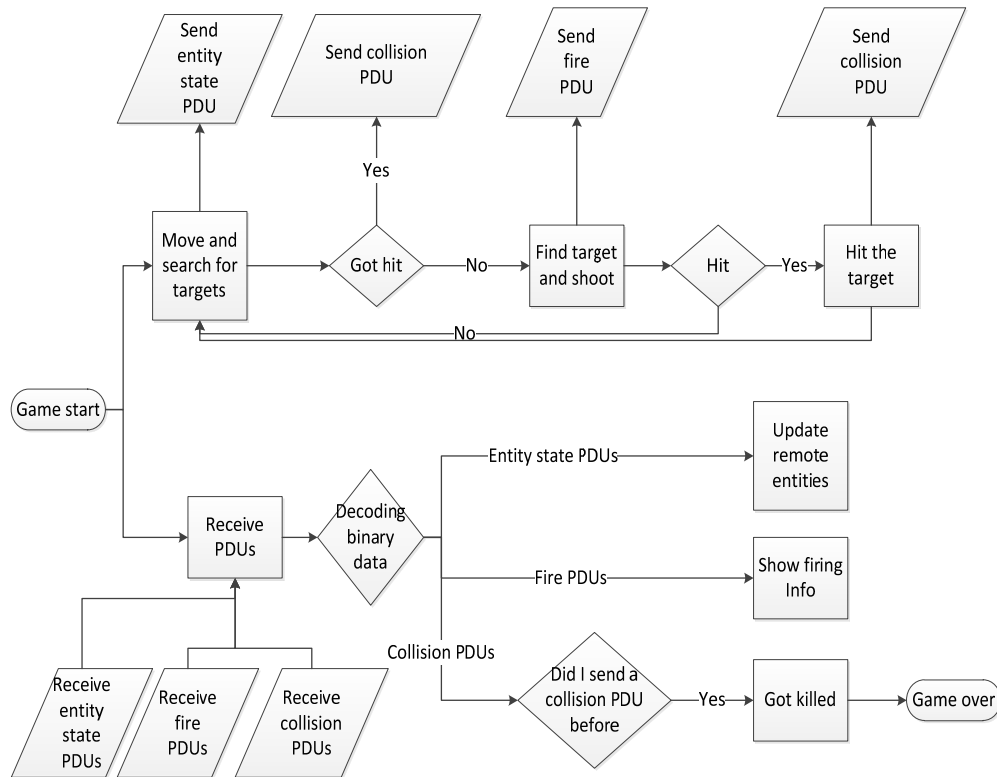


Figure 4. Tank game design.

C. PERFORMANCE IN DIFFERENT BROWSERS

Modern browsers such as Chrome, Firefox, and Safari use different JavaScript engines to execute JavaScript. Chrome uses V8 as its JavaScript engine. V8 is an open-source and high-performance JavaScript engine that is written in C++ and implements ECMAScript as specified in ECMA-262, 5th Edition (“Chrome V8,” n.d.). V8 can be run on most modern operating systems such as Windows (XP or newer), Mac OS X (10.5 or newer) and Linux systems. Firefox’s JavaScript engine is called SpiderMonkey, which is written in C/C++ (“Firefox SpiderMonkey,” n.d.). The latest JavaScript just-in-time (JIT) compiler for SpiderMonkey is called IonMonkey, which is implemented in the latest Firefox browser and can be installed in most modern operating systems. Safari uses

another JavaScript engine Nitro, but this thesis did not use Safari for experiments because Safari does not support WebRTC yet.

Different JavaScript engines cause variations in browser performance. There are many web applications for benchmarking JavaScript performance among different versions of browsers or different brands of browsers, including SunSpider, Kraken, and Octane. The benchmarking tests are varied, and it is hard to measure the performance precisely because there are too many variables (e.g., CPU, memory, browser version) to affect the benchmarking results. The common benchmarking tests include OS kernel simulation benchmark, 3D ray tracer, cryptography test, code decompression, PDF reader implementation, etc. There is an existing website showing benchmarking comparisons between modern browsers within different operating systems, and the website visually displays the differences among those modern browsers (“Arewefastyet,” n.d.).

Currently, WebSocket is supported widely by almost every browser (Deveria, n.d.-c). WebRTC is supported by a few browsers, including Chrome, Firefox, and Opera. The global usage of Chrome, Firefox and Opera is 28.39%, 4.69% and 0.36%, respectively (Deveria, n.d.-b). Therefore, the comparison of performance between browsers mainly focuses on Chrome and Firefox in this thesis.

IV. IMPLEMENTATION

In order to implement experiments, four services were run on the server side: WebSocket gateway, web service, PeerServer and primary peer. The experimental environment can be divided into two parts: WebSocket and WebRTC. This thesis, however, used one computer to run these four services. Figure 5 shows the experimental environment for this thesis. The first part was client-server framework using WebSocket gateway server. The server, which was downloaded from the Open-DIS project website, provides web server and server-side implementation of WebSocket.

Second part was peer-to-peer framework using PeerServer. PeerServer only has the capability to help broker connections between peers, so the experiments in this thesis used the web server from WebSocket gateway server for users to download web content, WebGL elements, 3D models and JavaScript code. One of the features of PeerJS is using unique IDs to make peer connections, so the newly-joined peer has to obtain the IDs of others to create peer connections beforehand. The primary peer was a webpage that collected the peer IDs and distributed a list of live peers to all connected peers. Although every client has a data channel with the primary peer, they would not send any DIS packets to the primary peer. A connected peer list was the only distributed data under this data channel.

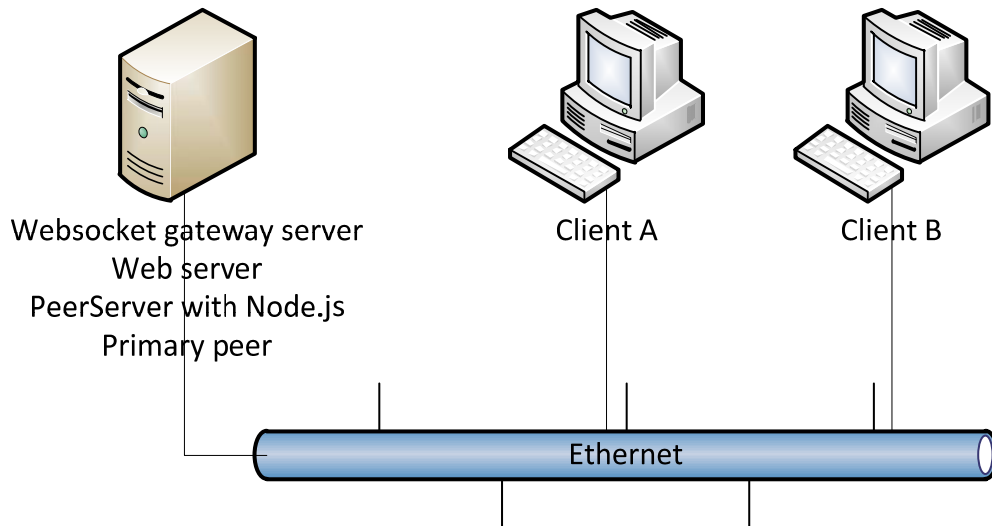


Figure 5. Experimental environment.

A. SERVER PREPARATIONS

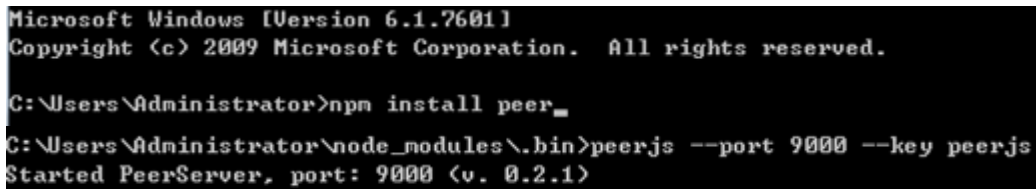
The following section describes preparatory works for the above-mentioned four services that were used to support the performance tests in this thesis.

1. WebSocket Gateway Server and Web Server

WebSocket gateway server ran on a computer with NetBeans installed. This server also had the capability of web server, so clients could get web content from a server IP address on port 8282. Port 8282 is a default setting for this server. This server also can receive native DIS messages that were broadcast from other simulation systems using UDP socket with port 3000, but this function was turned off manually in the experiments. The WebSocket was client-server architecture, so the expression in Figure 5 was that Client A sent DIS messages to WebSocket gateway server and then the server distributed the receiving DIS messages to Client B, and vice versa.

2. WebRTC: PeerServer and Primary Peer

PeerServer is Node.js-based application, so the server computer must install Node.js before running PeerServer. Node.js is a V8 (JavaScript engine) based platform for developing fast and scalable network applications. Figure 6 shows how to install and execute PeerServer after having Node.js installed. Node.js can be obtained from its official website: <http://nodejs.org/>. PeerServer used port 9000 to help broker connections between peers. The primary peer was one of PeerServer clients, so the primary peer could be executed after PeerServer was on. WebRTC is peer-to-peer connection with a rendezvous server; the framework is shown in Figure 3.



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Administrator>npm install peer_
C:\Users\Administrator\node_modules\.bin>peerjs --port 9000 --key peerjs
Started PeerServer, port: 9000 (v. 0.2.1)
```

Figure 6. Install and execute PeerServer.

B. BROWSER INITIATIVE PROCESSES AND BEHAVIORS

Once the server and services were established, client browsers could connect to the server to receive web contents. The web contents helped the browsers initiate a virtual world inside the window and interact with users. The initiative processes included importing necessary libraries; checking browser brands, and creating a canvas, scene, virtual objects, etc. Once the browser was initiated, it began sending and receiving DIS PDUs via WebSocket or WebRTC data channels. At the same time, the browser was ready to interact with users via the computer keyboard. The following describes the processes of initiating web browsers and the behaviors of exchanging DIS packets and interacting with users.

1. Import Libraries

The “dis.js” archive is an essential library for experiments in this thesis, and it can be downloaded from the Open-DIS project. It includes all DIS PDU classes and methods that convert from DIS PDU objects into binary format DIS and vice versa.

a. WebSocket

WebSocket is a native capability in web browsers that support WebSocket, so no specific library needs to be imported for WebSocket.

b. WebRTC

WebRTC is included in the web browser, and PeerJS wraps the browser’s WebRTC implementation. This thesis used PeerJS as an API to create WebRTC data channels, so the “peer.js” library obtained from the PeerJS official website must be imported at the beginning.

c. WebGL

WebGL is native in browser. This thesis applied “three.js,” which is layered on top of WebGL as a library for 3D graphics in web browsers (Parisi, 2012). It can be used to create scene graphs, camera, light, skybox, and basic 3D graphics, and it also can manipulate imported 3D models by applying different materials or shaders.

d. Others

Other libraries included OrbitControls.js as well as various loader and self-defined js archives. OrbitControls.js was used to control the camera by mouse in browsers, and the different loaders were for loading different formats of 3D models such as obj, js, mtl, and max. Figure 7 shows the relationships of importing libraries and other JavaScript programs.

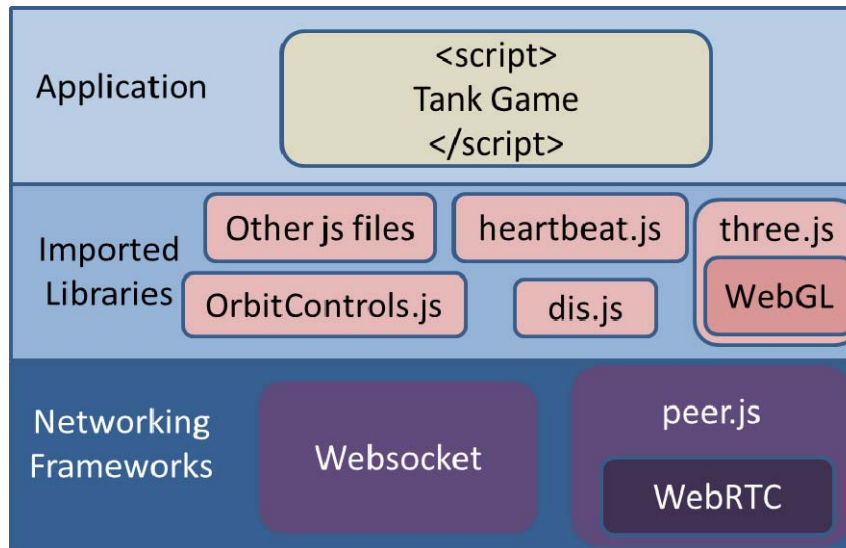


Figure 7. Relationships of importing libraries and other JavaScript programs.

2. Checking Web Browser Brand and Initiating WebSocket or WebRTC

Different browsers have different engines to implement JavaScript, so when an end user downloads web content from a web server, JavaScript code must check what kind of browser is being used. WebSocket and PeerJS that implements WebRTC have different ways to check browser brands.

a. *WebSocket*

WebSocket is widely supported by many browsers. The main browsers this thesis had to differentiate were Chrome and Firefox. Figure 8 shows the JavaScript code for distinguishing browser brands and establishing a connection to the server.

```

if(window.WebSocket)
    websocket = new WebSocket("ws://websocket.example.com:8282");
else if(Window.MozWebSocket)
    websocket = new MozWebSocket("ws://websocket.example.com:8282");
else
    alert("This web browser does not support web sockets");
  
```

Figure 8. Creating WebSocket code.

b. WebRTC

The WebRTC is wrapped by the PeerJS API, and it automatically checks browsers for web application developers. To initiate a peer object, a user has to randomly produce a peer ID to signal to the PeerServer, and then wait for a connection from other peers. If the ID is not given, the PeerServer will generate one for this client. Figure 9 shows the code for creating a peer object and signaling to PeerServer.

```
peer = new Peer( peerID, {host: PeerServer.example.com, port: 9000 });
```

Figure 9. Initiate a PeerJS client.

There are two situations for creating peer connections: pre-known the ID of prospective peer and unknown the ID of prospective peer. Figure 10 shows the architecture and sequence of creating connections among peers for performance tests in this thesis. The following three steps describe the mechanism of creating peer connections in the situation of an unknown prospective peer ID. This thesis uses the primary peer to help distribute an ID list to all connected peers. If the peer ID, however, is pre-known, a peer can create a peer connection to another peer directly with the help of PeerServer. For example, in Figure 10 the game client A creates a peer connection to primary peer because client A has known the ID of primary peer beforehand. Only the following step 1 and step 2 are involved in this case.

Step 1: The primary peer maintains a list of all the peers connected in the tank game. It must be running before any game clients start. When game clients begin execution, they contact the primary peer and provide their IDs. The Primary Peer in turn informs the game clients of the IDs of other peers. This allows the game clients to establish pair-wise connections for communication.

Step 2: When the first game client starts, it first contacts the PeerServer to get the information necessary to establish a connection to the Primary Peer. It then connects the Primary Peer and provides its ID.

Step 3: When a second game client starts it also first contacts the Peer Server as the first step for contacting the Primary Peer. It contacts the Primary Peer and provides its ID, and is in turn informed of the IDs of all other game clients. All game clients are informed of the ID of the new game client. All game clients can then establish pairwise connections between each other. The tank game client A established a WebRTC connection to game client B, and game client B establishes a connection to game client A.

In reality, each connection is a bidirectional channel. For example, if client A established a connection with client B, client B could use this connection to exchange data with client A. This thesis, however, constructed two data channels between each pair of clients for convenient configurations of sending and receiving behaviors.

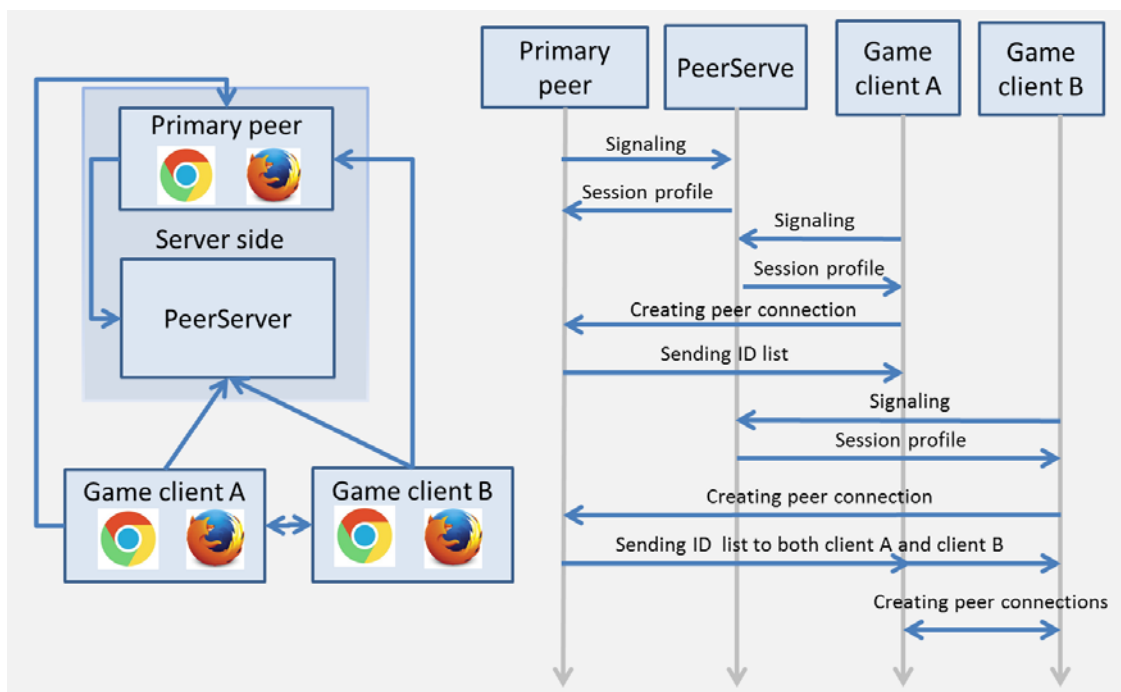


Figure 10. Architecture and sequences of creating peer connections.

3. Create Canvas, Scene, Camera, and Own Entities

HTML5 has a “canvas” element for drawing 2D and 3D graphics on web pages, and the three.js library provides methods to create and display animated 3D computer graphics on browsers that support WebGL. Figure 11 shows the codes to create a canvas and Figure 12 shows the basic elements in the scene. Every computer graphical elements—such as 3D graphics, light and camera—were added into the scene in the tank game.

At the same time, a controllable tank model was loaded by the JSON loader after the scene was created, and two JavaScript objects were created. One was to contain this controllable tank and an ESPDU that represents the states of this tank. Another object comprised a tank’s ammunition mesh and a different ESPDU that restored information of this ammunition. Figure 13 shows codes of JavaScript objects that contained the tank object and its corresponding ESPDU. It also shows some numbers were assigned in the field of entity type. These numbers were used to identify military hardware, and they referred to a SISO document called the “Enumeration and Bit Encode Values” (EBV). The EBV document is a long listing of standardized enumeration for simulation interoperability (*Simulation Interoperability Standards Organization (SISO) Reference for: Enumerations for Simulation Interoperability*, 2013).

```
<body>
  <div id="container">
    <canvas style="width:100%;height:95%;border:1px gray solid;">
  </div>
</body>
```

Figure 11. Creating canvas in a web page.

```

var mycanvas = document.getElementsByTagName("canvas")[0];
var w = mycanvas.clientWidth;
var h = mycanvas.clientHeight;
scene = new THREE.Scene();
camera = new THREE.PerspectiveCamera(
    30, // Field of view
    w / h, // Aspect ratio
    0.1, // Near
    10000 // Far
);
scene.add(camera);

var light = new THREE.PointLight( 0xFFFFFF );
scene.add( light );

var ambient = new THREE.AmbientLight( 0x222222 );
scene.add( ambient );

```

Figure 12. Creating scenes and adding graphical elements.

```

var ourEntity = {};
var loader = new THREE.JSONLoader();
ourEntity.tank = new Tank(loader);
ourEntity.lastEspdu = new dis.EntityStatePdu();
ourEntity.lastEspdu.entityType.entityKind = 1; //Platform
ourEntity.lastEspdu.entityType.domain = 1; //Land
ourEntity.lastEspdu.entityType.country = 225; //U.S.
ourEntity.lastEspdu.entityType.category = 1;
ourEntity.lastEspdu.entityType.subcategory = 1;
ourEntity.lastEspdu.entityType.spec = 3;
ourEntity.lastEspdu.entityID.entity = Math.round(Math.random() * 20000);

```

Figure 13. Creating tank meshes and ESPDUs in ourEntity objects.

4. Game Control and Graphics Rendering

Human-in-the-loop simulations must contain input devices for users to give orders, and all the web applications are run on computers and mobile devices. The development of this thesis's demonstration was based on using the keyboard and mouse as input devices. Players used the keyboard to control the

tank, fire missiles, and switch cameras; they used the mouse to change player perspective views. JavaScript provides corresponding key codes that are associated with keyboard characters for web application developers to develop different functions in different keys (Lautenschlager, n.d.). JavaScript also provides event handlers for keydown and keyup events in the window. The example tank game used keys “W” and “S” to move the tank forward and backward, keys “A” and “D” to turn the tank left and right, keys “Q” and “E” to adjust the tank’s barrel, key “Space” to fire a missile, and keys “1” and “2” to switch cameras. Figures 14 and 15 show examples of using JavaScript key codes to program different actions.

```
function onKeyDown(evt)
{
    switch (evt.keyCode)
    {
        case 49: // '1' // camera1
            camera = camera1; break;
        case 50: // '2' // camera2
            camera = camera2; break;
        case 65: // 'a'
            ourEntity.tank.rotationSpeed = +1.0; break;
        case 68: // 'd'
            ourEntity.tank.rotationSpeed = -1.0; break;
        case 87: // 'w'
            ourEntity.tank.translationSpeed = +10.0; break;
        case 83: // 's'
            ourEntity.tank.translationSpeed = -10.0; break;
        case 81: // 'q' // elevate barrel
            ourEntity.tank.elevateSpeed = 0.5; break;
        case 69: // 'e' // elevate
            ourEntity.tank.elevateSpeed = -0.5; break;
        case 32: // spacebar // Fire
            if(ourMunitionEntity.munition.missileReadyForShooting)
            {
                munitionFire = true;
                munitionSendCollisionPdu = true;
                ourMunitionEntity.munition.missileShoot = true;
            }
            break;
    }
};
```

Figure 14. Function of keydown events.

```

function onKeyUp(evt)
{
    switch (evt.keyCode)
    {
        case 65: // 'a'
            ourEntity.tank.rotationSpeed = 0.0; break;
        case 68: // 'd'
            ourEntity.tank.rotationSpeed = 0.0; break;
        case 87: // 'w'
            ourEntity.tank.translationSpeed = 0.0; break;
        case 83: // 's'
            ourEntity.tank.translationSpeed = 0.0; break;
        case 81: // 'q'
            ourEntity.tank.elevateSpeed = 0.0; break;
        case 69: // 'e'
            ourEntity.tank.elevateSpeed = 0.0; break;
    }
};
window.onkeydown = onKeyDown;
window.onkeyup = onKeyUp;

```

Figure 15. Function of keyup events and window event handlers.

The frequency of rendering 3D graphics affects the fluidity of the game. Most video games use 30 frames per second (fps) or 60 fps and some, such as Halo3, are locked at 30 fps maximum (“Frame rate,” n.d.). JavaScript provides a `setInterval` method that repeatedly calls a function in a specific interval. In the tank game, the rendering frequency was set to ten milliseconds, which equals to 100 fps, and all the game events and animations were based on that timestamp.

5. Convert Coordinate Systems

The DIS uses the ECEF (Earth-centered, Earth-fixed) coordinate system, which defines point (0, 0, 0) as the center of the earth. The positive x-axis is defined as running from this point out to where the equator and the Prime Meridian intersect. The positive y-axis runs from the center point out to where the equator intersects the 90 degree east meridian, and the positive z-axis points from the center toward the North Pole. The “dis.js” library provides methods to convert coordinates between ECEF and ENU (east, north and up), which is a set

of local coordinates with a given geodetic point. Figure 16 presents the coordinate systems of ECEF and ENU. The “dis.js” also helps convert between ECEF and latitude, longitude and altitude.

The tank game is a self-interactive game, which means players are always in the same local coordinate system. The game, however, still converts its local coordinates from ENU to ECEF in case other DIS simulations want to communicate with it in the future. Besides the conversion of location, the entity’s orientation must also be converted. Rotation in three.js is based on quaternion, which uses four numbers to represent an entity’s orientation in a 3D world. On the other hand, ESPDU only provides three fields that are Euler angles to express orientation. There are many examples of conversion between quaternion and Euler angles on the Internet. Figure 17 is an example of a conversion between coordinates.

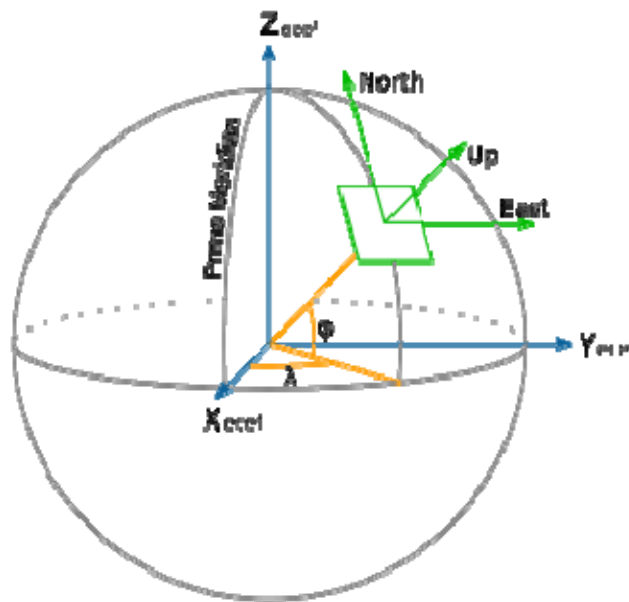


Figure 16. Earth centered, earth fixed; and east, north, up coordinates
("Axes conventions," n.d.)

```

var rangeCoordinates = new dis.RangeCoordinates(0.0, 0.0, 0.0);

// convert from local coordinate system to global coordinate system.
var localToGlobal = function (entity)
{
    var entityPosition = {x:entity.tank.mesh.position.x,
                        y:entity.tank.mesh.position.y, z:entity.tank.mesh.position.z };
    var toGlobalCoordinates = rangeCoordinates.ENUObjectToECEF(entityPosition);
    entity.lastEspdu.entityLocation.x = toGlobalCoordinates.x;
    entity.lastEspdu.entityLocation.y = toGlobalCoordinates.y;
    entity.lastEspdu.entityLocation.z = toGlobalCoordinates.z;

    //orientation
    var pitchYawRoll = quatToEuler(entity.tank.mesh.quaternion);
    entity.lastEspdu.entityOrientation.theta = pitchYawRoll.y;
};

//convert from global coordinates to local coordinate system.
var globalToLocal = function(mesh, espud)
{
    var localCoordinates = rangeCoordinates.ECEFObjectToENU(espud.entityLocation);
    mesh.position.x = localCoordinates.x;
    mesh.position.y = localCoordinates.y;
    mesh.position.z = localCoordinates.z;

    //orientation
    mesh.rotation.y = espud.entityOrientation.theta;
};

```

Figure 17. Conversion between ECEF and ENU.

6. Send DIS Packets

DIS simulation uses heartbeat strategy that periodically sends DIS packets to update entity information and maintain entity existence; the “dis.js” library provides methods to convert every DIS PDU from JavaScript object into binary format DIS messages. In the demonstration tank game, entity state PDUs were sent periodically to the server or other PeerJS clients.

Collision PDU and fire PDU, on the other hand, are event-oriented. Collision PDUs were issued when there was a collision event between a tank missile and opposing tank. Fire PDUs were issued whenever someone was

shooting. The periodic sending of DIS PDUs also helped maintain entity existence in other clients, because each client browser would check the receiving time for every entity. If an entity's last receiving time was 3 seconds ago, which was a game setting, this entity would be removed from client browsers. Figure 18 shows example code of a heartbeat function in setInterval method, and a trimmed DIS message that was transferred via the WebSocket.

```
var heartbeat = function()
{
    var dataBuffer = new ArrayBuffer(1500);
    var outputStream = new dis.OutputStream(dataBuffer);
    ourEntity.lastEspdu.encodeToBinaryDIS(outputStream);
    var trimmedData = dataBuffer.slice(0, outputStream.currentPosition);
    websocket.send(trimmedData);
};

// Periodically send heartbeat messages
setInterval(heartbeat, 10);
```

Figure 18. Heartbeat function.

PeerJS uses a similar method to send DIS messages, but the sending mechanism is via PeerJS DataConnection object. Figure 19 shows how to create a DataConnection object from a peer object, and the sending method. The trimmedData is the same with WebSocket.

```
var connectTo = peer.connect('other peerID');

connectTo.send(trimmedData);
```

Figure 19. PeerJS sending method.

7. Receive and Decode DIS Messages

The codes used to receive data are different between WebSocket and PeerJS. The decoding processes, however, are the same.

a. *Receive by WebSocket*

Before receiving data from WebSocket, the format to receive binary messages and attach functions for various events must be set. Figure 20 shows the code for setting the WebSocket. `WebSocket.onmessage` is the function used to handle receiving data from the server site.

```
websocket.binaryType = 'arraybuffer';

websocket.onopen = function(evt){console.log("websocket onopen");};
websocket.onclose = function(evt){console.log("websocket close");};
websocket.onerror = function(evt){console.log("websocket error");};
websocket.onmessage = function(evt){
// Handle received DIS messages here
};
```

Figure 20. Set format and attach functions.

b. *Receive by PeerJS*

After creating a peer object, developers have to set listeners for peer events. The main method this thesis used was “`peer.on,`” and the listening event was “`connection,`” with a function to handle the incoming messages. This event will receive a `dataConnection` object, which wraps WebRTC’s `DataChannel`, and pass this object to the handling function. Figure 21 shows the code to listen for events.

```
peer.on('connection', function(dataConnection) { ... });
```

Figure 21. PeerJS event listener.

The `DataConnection` class contains three methods: “send,” “close” and “on.” The “send” method sends data to the remote peer, and “close” closes the data connection and cleans up underlying `DataChannels` and `PeerConnections`. The “on” method can be set for listening for data connection events: “data,” “open,” “close” and “error.” The thesis experiments used the “data” event that is emitted when data is received from remote peers to receive and handle DIS messages. Figure 22 shows the example code.

```
peer.on('connection', connect);

function connect(connReceived) {
  var peerId = connReceived.peer;
  console.log('remote peer id: ', peerId);

  connReceived.on('data', function(data) {
    //Handle received DIS messages here
  });
  connReceived.on('close', function() {
    delete connectedPeers[peerId];
  });
}
```

Figure 22. Example code of receiving events function.

c. ***Decoding DIS Messages***

The mechanism of decoding DIS packets is the same in `WebSocket` and `WebRTC` because both rely on the “dis.js” library to interpret DIS messages. The function of the `WebSocket` and the `WebRTC` is to send and receive application data. The first step in interpreting DIS messages is to allocate packets to different types of PDUs using a method called `dis.PduFactory()`. Different PDUs contain different information with different data lengths. Every PDU, however, has the same PDU header, which has 96 bits to store basic information such as protocol version (8-bit enumeration), exercise ID (8-bit unsigned integer), and

PDU type (8-bit enumeration). The `dis.PduFactory()` method uses the PDU type, which is the third byte in the PDU header, to distinguish what kind of PDU should be assigned. The demonstration game in this thesis used three different types of PDUs: entity state PDU, collision PDU, and fire PDU. The respective lengths of these three DIS PDUs are 1152 bits, 480 bits and 768 bits, and the respective enumeration numbers of these three PDUs are 1, 4, and 2. Figure 23 shows how to use `dis.PduFactory()`.

```
var pduFactory = new dis.PduFactory();  
var newPdu = pduFactory.createPdu(evt.data);//Websocket  
var newPdu = pduFactory.createPdu(data);//PeerJS
```

Figure 23. Example code of `dis.PduFactory()`.

After differentiating the PDU type, it is time to deal with receiving the DIS messages: entity state PDU, collision PDU, and fire PDU. ESPDU is used to update entity states in client browsers. Every client would create a JavaScript object, which is used to record all remote entities from other game participants. If the receiving ESPDU's entity ID did not previously exist, client browsers will recode the new ESPDU in an all-remote-entity object, and create a corresponding 3D model loaded by JSON loader to represent this entity ID. If there is a 3D model having the same entity ID corresponding with the receiving ESPDU, browsers will update this 3D model's states such as location, velocity, and orientation. Collision PDUs will be issued when a collision occurs between entities. In the demonstration tank game, collision PDUs were issued whenever a tank got hit; both the firing missile and the hit tank issued collision PDUs. Fire PDU in this scenario was for showing all firing events on the screen.

8. Miscellaneous

There are some miscellaneous things that have not been mentioned, but which are also very important for creating a browser-based DIS simulation. These include how to load meshes, how to detect collision in a virtual environment, how to ensure entity existence, and the application of dead reckoning.

a. Loading 3D Objects

Creating the 3D graphics took longer than creating a JavaScript object. When a client browser received an ESPDU whose entity ID was new to this client, the client would create a 3D model to represent this new entity. The 3D model, however, could not be located immediately in the virtual world before it was fully loaded. In the tank game example, a function continued updating entity states and the corresponding 3D model each time the clients received the same ESPDUs. Once the 3D model was fully loaded into the virtual world, an opposing tank would show in the browser. The following code describes a way to update entity and 3D model properties. Figure 24 shows the code of updating the entity's model and states.

```

var updateEntityMesh = function ()
{
  for(var e = 0; e < needUpdateEntityArray.length; e++ )
  {
    var needUpdateEntity = needUpdateEntityArray.shift();
    for(var eidString in allEntities)
    {
      var currentEntity = allEntities[eidString];
      if(eidString === needUpdateEntity)
      {
        if(currentEntity.munition)
        {
          if(currentEntity.munition.mesh)
          {
            if(currentEntity.lastEspdu.entityType.entityKind === 2)
            {
              globalToLocal(currentEntity.munition.mesh, currentEntity.lastEspdu);
              DeadReckoningParameterGlobalToLocal(currentEntity.munition, currentEntity.lastEspdu);
              allEntities[eidString] = currentEntity;
            }
          }
        }
        if(currentEntity.tank)
        {
          if(currentEntity.tank.mesh)
          {
            globalToLocal(currentEntity.tank.mesh, currentEntity.lastEspdu);
            allEntities[eidString] = currentEntity;
          }
        }
      }
    }
  }
};

```

Figure 24. Code of updating entity states and 3D model.

b. Entity Collision Detection

The “three.js” library provides methods for ray casting that can be used to detect collision in a virtual world. The way of using ray casting is to push all prospective collided objects into an array before doing ray casting. In the demonstration game, ray casting was used for checking collisions between: tank and terrain, tank and tank, and tank and skybox. Tanks cast a ray to minus the y-axis (i.e., toward the ground), so that they could rise and fall on (i.e., “follow”) the terrain by having a collision between tank and terrain. Tanks also cast rays in another four directions—front, rear, right, and left—for detecting other tanks and any vertical terrain. Ray casting in this game was used not only for collision

detection, but also for pointing out missile directions: shooting a ray from a turret to find a collision point, and then using this point to determine projectile path by computing between colliding point and turret.

c. *Checking Existence*

The DIS is a heartbeat-based simulation, so every client should receive ESPDUs periodically from other clients. There is a field in the PDU header for timestamp. This field can be used to record the last time heard from other ESPDUs. If a simulation participant did not receive ESPDUs from remote peers in a period-of-time, the participant removed 3D models and properties that belonged to those remote peers. There were six elements removed when the client browser did not hear any ESPDU from a specific peer: tank mesh and ammunition mesh in the scene, tank, and ammunition properties in the all-remote-entity object, and tank and ammunition objects in the collision array.

d. *Dead Reckoning*

Dead reckoning is used to predict an entity's next position by using the entity's current position with a dead-reckoning algorithm, linear acceleration, angular velocity, and other parameters when the entity does not receive the next prospective ESPDUs to update its position. Dead reckoning can be utilized to cover an entity's stutters caused by the heartbeat period. The demonstration tank game used dead reckoning for a projectile's movement to reduce visual jumping, in-browser, of other game participants. A projectile's movement with dead reckoning that complements intervals between two consecutive ESPDUs also increases the accuracy of issuing collision PDUs, because the heartbeat strategy might cause the projectile to "jump" over an opponent's tank without any collision or intersection.

V. PERFORMANCE TESTS

This thesis investigated two networking frameworks, WebSocket and WebRTC, and used these two frameworks in performance experiments and the resulting data. The WebSocket was constructed as client-server architecture. In contrast, the WebRTC was based on peer-to-peer architecture after a data channel was constructed. This thesis also incorporated WebGL components as a comparable factor, which compared the presences of WebGL components in the browser. WebGL enabled the browser's rendering of 2D and 3D graphics, but it also increased CPU and GPU loadings by computing graphics transformation, scaling, rotation, translation, etc. Furthermore, different browsers used different JavaScript engines, so the differences between the browsers (Chrome version 36.0.1985.143m and Firefox version 24.7.0) were compared. Measuring tools for this thesis included self-created functions, Chrome developer tools, and Firefox developer tools.

A. SENDING AND RECEIVING ABILITY

To measure the sending performances in different networking framework and browsers, this thesis created a function to send ESPDU packets. The receiving numbers were increased when the onmessage function was called and packets were received by the WebSocket. The PeerJS received PDUs when the dataConnection object heard 'data' events. Figure 25 shows the measureDIS function for sending DIS packets in WebSocket. The measurement evaluated how much time was needed for sending 10,000 DIS packets, with millisecond as the time unit. The function had a 'sendercounter' variable to cumulate the total number of sending. At the same time, this function also converted the total number to a serial number for dropping test into the entityAppearance field in ESPDU. PeerJS used connectTo (a dataConnection object) to send the trimmed data (see Chapter IV, Section 6, Send DIS Packets).


```

var measureDIS = function()
{
    ourEntity.lastEspdu = new dis.EntityStatePdu();
    startTime = new Date();
    senderCounter = 0;
    while(senderCounter < 10000)
    {
        ourEntity.lastEspdu.entityAppearance = totalSend+1;
        var dataBuffer = new ArrayBuffer(1500);
        var outputStream = new dis.OutputStream(dataBuffer);
        ourEntity.lastEspdu.encodeToBinaryDIS(outputStream);
        var trimmedData = dataBuffer.slice(0, outputStream.currentPosition);
        websocket.send(trimmedData);
        senderCounter ++;
        totalSend ++;
    }
    endTime = new Date();
    var elapsedTime = endTime.getTime() - startTime.getTime();
    console.log("time:", elapsedTime, " send packets:", senderCounter);
};

```

Figure 25. MeasureDIS function for sending DIS packets.

Figure 26 shows the example of a WebSocket receiving DIS packets. The measurement was the same with the sending function that measured how much time was needed for receiving 10,000 PDUs. Figure 27 shows the method that the PeerJS used to receive data.

```

websocket.onmessage = function(evt)
{
    receiveCounter++;
    totalReceive ++;
    if(receiveCounter === 1){startTime = new Date();}
    if(receiveCounter % 10000 === 0)
    {
        var endTime = new Date();
        var elapsedTime = endTime.getTime() - startTime.getTime();

        console.log("time:", elapsedTime, "receive packets:", receiveCounter);
        receiveCounter = 0;
        startTime = new Date();
    }
};

```

Figure 26. Measuring function of receiving DIS packets in WebSocket.

```

connReceived.on('data', function(data)
{
    //same with websocket onmessage.
    receiveCounter++;
    totalReceive ++;
    if(receiveCounter === 1){startTime = new Date();}
    .....
});

```

Figure 27. Measuring function of receiving DIS packets in PeerJS.

The following section shows the transformed outcomes of running the above functions 20 times based on a one-on-one situation, one sender, and one receiver. As a result, 20 runs of a specific time were spent on sending and receiving 10,000 PDUs. The thesis used MS Excel to convert the results into the number of Entity State PDUs that were sending and receiving per second, and used JMP for statistics.

Figure 28 shows the outcome of pure sending and receiving rates in different combinations of browsers. The left eight experiments are receivers, and the remaining experiments are senders. The green lines are the means of each experiment, and the blue bars can be used to visually compare the statistical difference between any pair of experiments. If a pair of bars overlaps from each other, these two experiments are not significantly different, and vice versa. CC means that the Chrome browser sent to the Chrome browser; CF means the Chrome browser sent to the Firefox browser, and so on. Figure 28 shows that senders were overwhelmingly faster than recipients, because the sending rates were counted by executing the measureDIS function instead of actually sending out to the browsers. In addition, none of the experiments found message losses in the WebSocket framework, and rarely were the DIS packets lost in the WebRTC framework. The sending data was not sent out by browsers; instead, it was queued in the senders' memory. For example, if a sender's browser invoked a function that used WebSocket to send 10,000 ESPDUs per second, but the receiver only got 7,000 PDUs per second; the remaining 3000 ESPDUs per

second would buffer in the sender's memory. Hence, the performances of sending and receiving DIS messages in different frameworks in different browsers should refer to the receiving throughputs.

Figure 29 focuses on the DIS receiving rates, and shows that the WebSocket had relatively stable receiving rates (around 7,000 to 8,000 per second in different browsers). According to the WebSocket framework—which was client-server architecture—the recipients received DIS packets from the WebSocket gateway server, which means the receiving performances are dependent on the server's capability. In contrast, the receiving abilities of PeerJS, which implemented WebRTC and were based on peer-to-peer connections, were varied. The recipients received around 2,000 per second in Chrome sent to Chrome and Chrome sent to Firefox, but had better performance of around 11,000 to 13,000 per second in Firefox sent to Chrome and Firefox sent to Firefox.

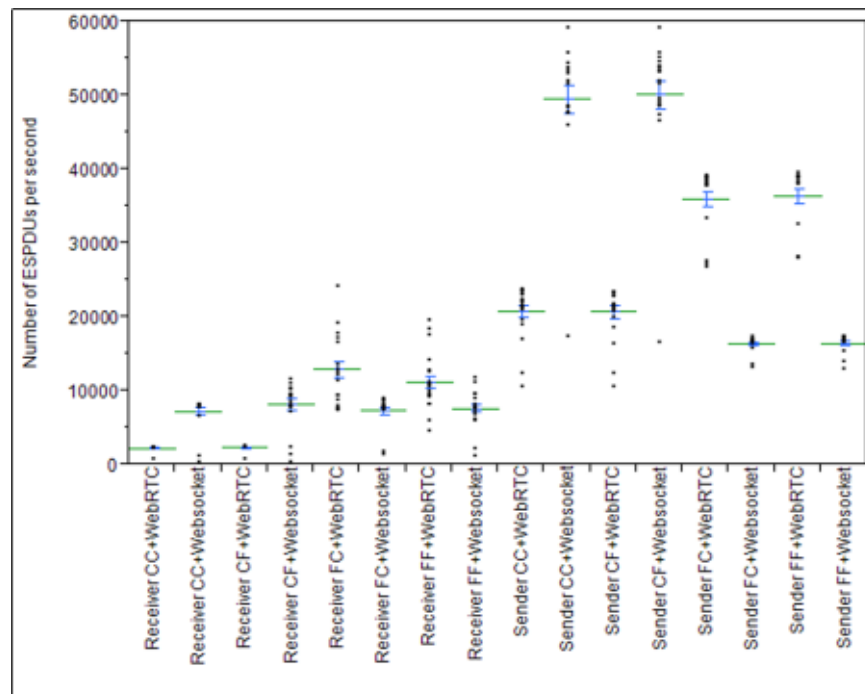


Figure 28. Purely sending and receiving capabilities.

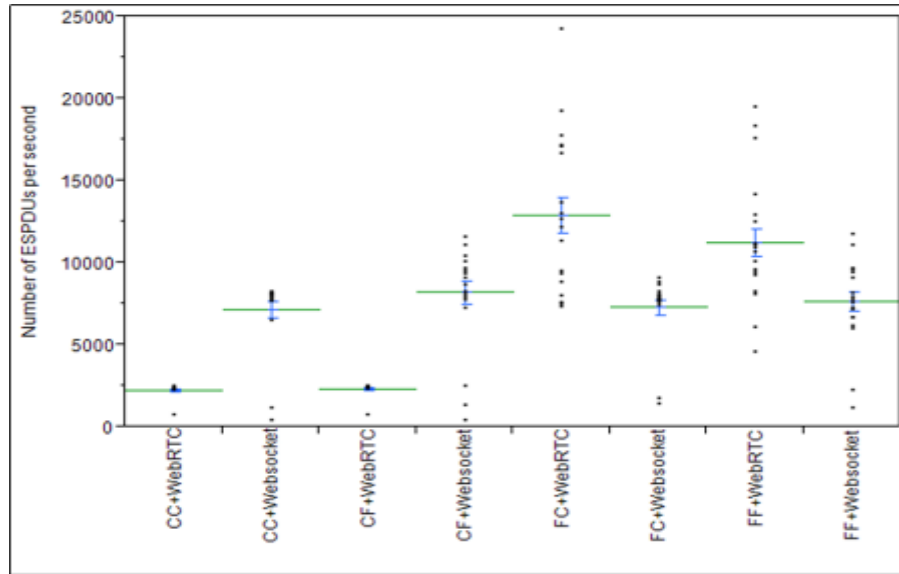


Figure 29. Purely receiving capabilities.

According to the above conclusions, sending rates did not correspond to the receiving rates, so the performance with WebGL elements should focus on the receiving capabilities. Figure 30 shows comparisons between the presences of WebGL, and reveals that these were graphically different among the mean lines. It seems like that WebGL had some level of influence on receiving the DIS messages, because the receiving capabilities with WebGL elements are lower than without WebGL elements.

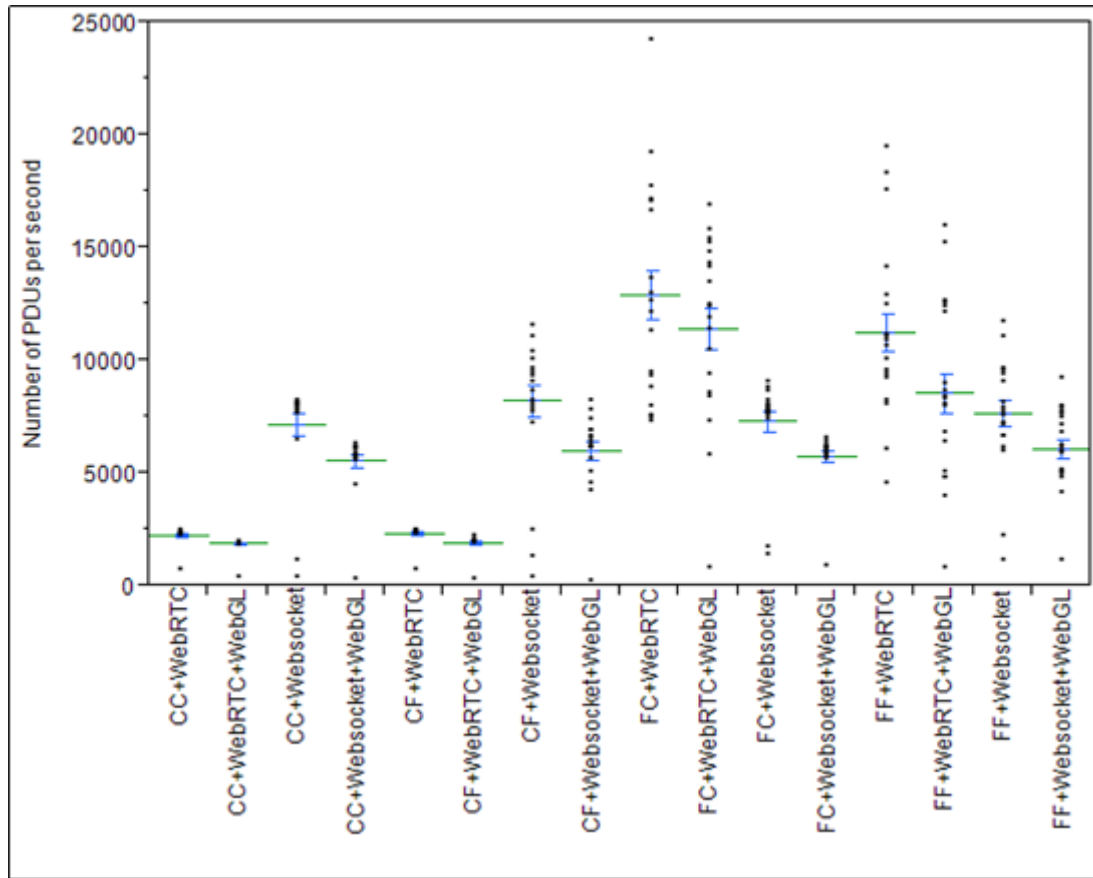


Figure 30. Comparisons of receiving capabilities between the presences of WebGL.

To understand the differences among receiving rates, this thesis used a student's t-tests to find out if pairs of receiving rates were significantly different from one another. By looking up rates in the table of t-distribution, the p-value could be found to compare with the alpha level. This thesis used 0.01 as the alpha. If the p-value greater than the alpha, there was no significant difference between the comparing pair, and vice versa. The p-value only indicated the degree of difference; it did not indicate any better performance. Table 2 shows the performances of that the WebSocket framework versus the WebRTC framework was significantly different, because all the p-values are less than 0.0001. The differences can be checked graphically in Figure 29.

Comparison between				P-value
experimental combination	average number of receiving PDUs	experimental combination	average number of receiving PDUs	
CC+WebSocket	7203.79	CC+WebRTC	2272.54	<.0001
CF+WebSocket	8212.41	CF+WebRTC	2300.51	<.0001
FC+WebRTC	12917.75	FC+WebSocket	7311.62	<.0001
FF+WebRTC	11239.30	FF+WebSocket	7695.37	<.0001

Table 2. t-tests of receiving capability in WebSocket and WebRTC frameworks without WebGL components. The experiment at left had the higher performance.

Table 3 presents paring comparisons between browsers using the WebSocket framework; all the p-values are greater than 0.01, which means there were no significant difference of receiving rates between browsers when using WebSocket framework. Greater p-value represents less difference between the comparing pair. The WebSocket framework had stable performances in both Chrome and Firefox browsers.

Comparison between				P-value
experimental combination	average number of receiving PDUs	experimental combination	average number of receiving PDUs	
CF+WebSocket	8212.41	CC+WebSocket	7203.79	0.2151
CF+WebSocket	8212.41	FC+WebSocket	7311.62	0.2681
CF+WebSocket	8212.41	FF+WebSocket	7695.37	0.5247
FF+WebSocket	7695.37	CC+WebSocket	7203.79	0.5453
FF+WebSocket	7695.37	FC+WebSocket	7311.62	0.6368
FC+WebSocket	7311.62	CC+WebSocket	7203.79	0.8944

Table 3. t-tests of receiving capability between browsers without WebGL components in WebSocket framework.

On the other hand, Table 4 shows paring comparisons between browsers using WebRTC framework. In all cases, Firefox is shown to send via WebRTC significantly faster than Chrome does. The p-value 0.9725 indicates that using

the Chrome browser to send DIS packets to the Chrome browser had similar performance when using the Chrome browser to send DIS packets to the Firefox browser. The p-value 0.0395 indicates that using the Firefox browser to DIS packets to the Chrome browser had similar performance as using the Firefox browser to send DIS packets to the Chrome browser, because the alpha level was 0.01. These results show that the same sender had similar receiving rates in recipient browsers.

Comparison between				P-value
experimental combination	average number of receiving PDUs	experimental combination	average number of receiving PDUs	
FC+WebRTC	12917.75	CC+WebRTC	2272.54	<.0001
FC+WebRTC	12917.75	CF+WebRTC	2300.51	<.0001
FF+WebRTC	11239.30	CC+WebRTC	2272.54	<.0001
FF+WebRTC	11239.30	CF+WebRTC	2300.51	<.0001
FC+WebRTC	12917.75	FF+WebRTC	11239.30	0.0395
CF+WebRTC	2300.51	CC+WebRTC	2272.54	0.9725

Table 4. t-tests of receiving capability between browsers without WebGL components in WebRTC framework.

The last pairing comparisons were the differences between the presence of WebGL materials (see Table 5). It is hard to decide whether the presences of WebGL materials affected the PDU sending and receiving performance, because there were two p-values that were less than 0.01. Most comparisons, however, were not significantly different from each other. Besides focusing on Table 5, the comparisons with less than 0.01 p-values had very good performances on sending and receiving DIS messages. Figure 30 shows that both FF+WebRTC+WebGL and CF+WebSocket+WebGL had the capability of sending and receiving more than 5,000 DIS packets per second, which was much faster than using the Chrome sent to Chrome and the Chrome sent to Firefox in the WebRTC framework.

Comparison between				P-value
experimental combination	average number of receiving PDUs	experimental combination	average number of receiving PDUs	
FF+WebRTC	11239.30	FF+WebRTC+WebGL	8562.47	0.0011
CF+WebSocket	8212.41	CF+WebSocket+WebGL	6008.55	0.007
CC+WebSocket	7203.79	CC+WebSocket+WebGL	5568.12	0.0448
FF+WebSocket	7695.37	FF+WebSocket+WebGL	6096.03	0.0497
FC+WebSocket	7311.62	FC+WebSocket+WebGL	5727.37	0.0519
FC+WebRTC	12917.75	FC+WebRTC+WebGL	11403.85	0.0632
CC+WebRTC	2272.54	CC+WebRTC+WebGL	1877.29	0.6267
CF+WebRTC	2300.51	CF+WebRTC+WebGL	1931.61	0.6499

Table 5. t-test between presences of WebGL components

B. PROFILING JAVASCRIPT PERFORMANCES

This thesis used Chrome developer tools and Firefox developer tools to profile experimental browsers. The profiling tools were used to record JavaScript performances in a period-of-time, and the recording data included percentages of time spent and explicit time spent for each function in this period-of-time. For example, if a user starts profiling then running the measureDIS function for two different lengths of time—one long and the other short—the measureDIS function would occupy a small percentage of the long period, but more in the short period. Figure 31 shows running the measureDIS function in a long period-of-time, and Figure 32 shows running the same function in a short period-of-time. The difference was through running the measureDIS function with a shorter time (39062.7 ms) in a short period; this function still captured more (19.06%) than the one that recorded for a long period-of-time (40247.1 ms with 10.15%). The ‘Self’ column indicates the time to complete the current function that excludes any functions it called. The ‘Total’ column shows the time to complete the current function and any functions it called (“Profiling JavaScript Performance,” n.d.).

Timeline Profiles Resources Audits Console NetBeans					
Tree (Top Down) ▼ 🔍 ✕					
Self ▼		Total		Function	
350839.9 ms	88.57 %	350839.9 ms	88.57 %	(idle)	
4575.4 ms	1.16 %	4575.4 ms	1.16 %	(program)	
449.8 ms	0.11 %	449.8 ms	0.11 %	(garbage collector)	
28.6 ms	0.01 %	40247.1 ms	10.16 %	▼ (anonymous function)	
1008.1 ms	0.25 %	40216.1 ms	10.15 %	▶ measureDIS	
2.4 ms	0.00 %	2.4 ms	0.00 %	⚠ dis.EntityStatePdu.encodeToBinaryDIS	
4.8 ms	0.00 %	4.8 ms	0.00 %	set name	

Figure 31. Running measureDIS function for a long period-of-time.

Timeline Profiles Resources Audits Console NetBeans					
Tree (Top Down) ▼ 🔍 ✕					
Self ▼		Total		Function	
153119.2 ms	74.72 %	153119.2 ms	74.72 %	(idle)	
12311.2 ms	6.01 %	12311.2 ms	6.01 %	(program)	
387.4 ms	0.19 %	387.4 ms	0.19 %	(garbage collector)	
29.1 ms	0.01 %	39094.9 ms	19.08 %	▼ (anonymous function)	
874.9 ms	0.43 %	39062.7 ms	19.06 %	▶ measureDIS	
3.2 ms	0.00 %	3.2 ms	0.00 %	dis.DeadReckoningParameter.encodeToBinaryDIS	

Figure 32. Running measureDIS function for a short period-of-time.

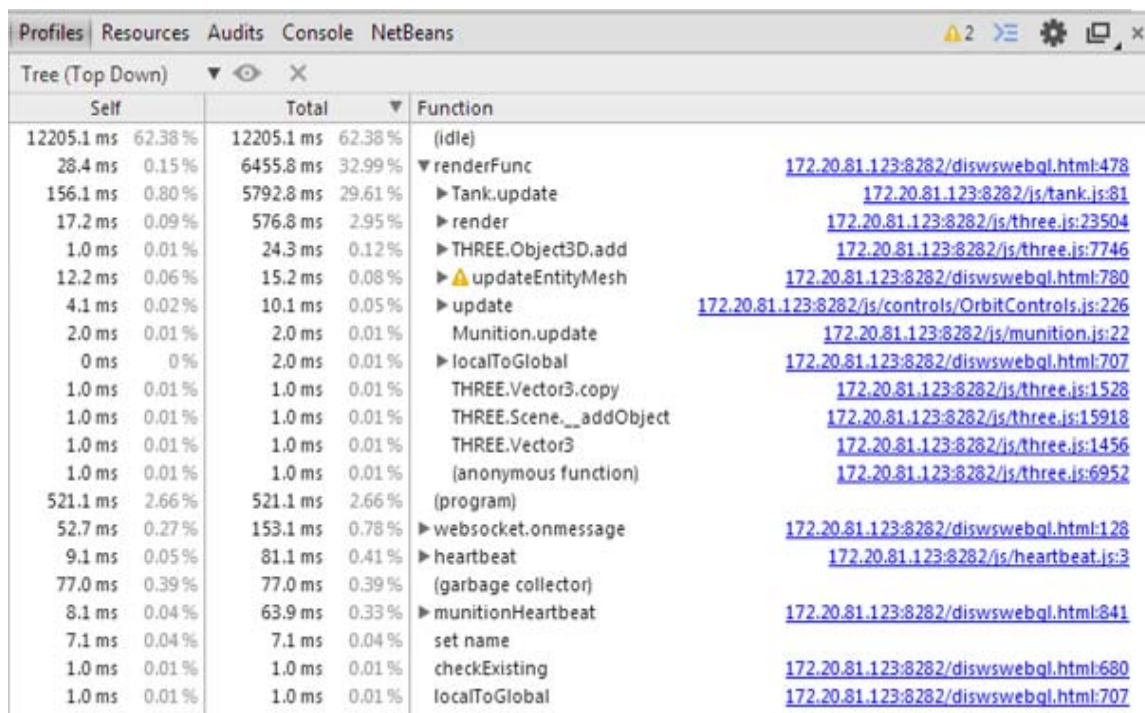
Firefox developer tools also have profiling tools. Furthermore, the package includes a function to select a certain section to analyze JavaScript performances, so the behaviors of sending and receiving DIS PDUs can be examined. Looking at the above results of sending and receiving DIS packet capabilities, however, the data interchanging rates should refer to the receiving capabilities; this means that, although the browser consumes all resources to execute the measureDIS function, the efficient outputs would not reflect on the receivers. Therefore, it is not worthwhile to check the JavaScript performance of purely sending DIS packets. Appendix B-1 is a profiling example of sending DIS PDUs in the WebSocket framework. Appendix B-2 selects a section that focused on the sending function. Appendix B-3 is the corresponding receiver in this example, which evenly consumed approximately 5–7% of the browser resource.

After understanding the characteristic of profiling tools, it is difficult to analyze a browser's behavior by profiling it with the measureDIS function, because the time to invoke the measureDIS function and the time to complete sending and receiving DIS messages varies in different networking environments and different combinations of browsers. Therefore, this thesis used the profiling tools to record the JavaScript performances in the demonstration tank game as a practical application, instead of executing the measureDIS function. Based on the above results, applying WebGL materials decreased PDU receiving capability in both the WebSocket and WebRTC frameworks, but the amount of receiving DIS PDUs remained at approximately 1,900 per second.

Figures 33 and 34 display the recording of the tank game for a period-of-time in different networking frameworks. The game setting was that each browser had a controllable tank and a projectile that could be fired every two seconds. The tank and projectile sent ESPDUs every ten milliseconds, which was different from the measureDIS function that sent 200,000 ESPDUs, successively. The fire PDU and collision PDU were sent when the corresponding events happened. The advantage of profiling the tank game is that it is easy to analyze the sending and receiving performances with WebGL elements in different networking frameworks, and in client-server versus peer-to-peer configurations. Because the profiling tools used the percentage of function executing time and most of the functions were invoked periodically (which included graphical rendering and DIS heartbeat functions), the different profiling times did not influence the results.

This tank game sent approximately 200 ESPDUs per second, which is below the receiving capabilities in any situation. Figure 33 is a profile of the tank game using the WebSocket framework. The profile showed that the renderFunc function consumed 32.99% of the recording time to execute this function. The renderFunc function is a major function in this tank game, used to render graphics and check collisions between 3D objects in the virtual world. It contained the tank.update function and many 'THREE' functions, which indicated the browser spent most of its resources on the WebGL materials. In addition, the

websocket.onmessage, heartbeat and munitionHeartbeat functions could be found near the bottom of the Function column. The respective percentages for these three functions are 0.78%, 0.41%, and 0.33% in the 'Total' column. These results showed that the exchange of DIS messages did not give browsers a heavy workload. Figure 34 is a profile of the tank game using the WebRTC framework, and it shows a similar performance with using WebSocket framework. The function of _dc.onmessage near the bottom of the Function column is similar to the function of websocket.onmessage in WebSocket for listening to incoming events. Firefox developer tools profiled similar results, which are shown in Appendix C-1 and C-2. C-1, used the WebSocket framework, and C-2 used the WebRTC framework.



Self		Total		Function	
12205.1 ms	62.38 %	12205.1 ms	62.38 %	(idle)	
28.4 ms	0.15 %	6455.8 ms	32.99 %	▼ renderFunc	172.20.81.123:8282/diswswebgl.html:478
156.1 ms	0.80 %	5792.8 ms	29.61 %	▶ Tank.update	172.20.81.123:8282/js/tank.js:81
17.2 ms	0.09 %	576.8 ms	2.95 %	▶ render	172.20.81.123:8282/js/three.js:23504
1.0 ms	0.01 %	24.3 ms	0.12 %	▶ THREE.Object3D.add	172.20.81.123:8282/js/three.js:7746
12.2 ms	0.06 %	15.2 ms	0.08 %	▶ ⚠ updateEntityMesh	172.20.81.123:8282/diswswebgl.html:780
4.1 ms	0.02 %	10.1 ms	0.05 %	▶ update	172.20.81.123:8282/js/controls/OrbitControls.js:226
2.0 ms	0.01 %	2.0 ms	0.01 %	Munition.update	172.20.81.123:8282/js/munition.js:22
0 ms	0 %	2.0 ms	0.01 %	▶ localToGlobal	172.20.81.123:8282/diswswebgl.html:707
1.0 ms	0.01 %	1.0 ms	0.01 %	THREE.Vector3.copy	172.20.81.123:8282/js/three.js:1528
1.0 ms	0.01 %	1.0 ms	0.01 %	THREE.Scene.__addObject	172.20.81.123:8282/js/three.js:15918
1.0 ms	0.01 %	1.0 ms	0.01 %	THREE.Vector3	172.20.81.123:8282/js/three.js:1456
1.0 ms	0.01 %	1.0 ms	0.01 %	(anonymous function)	172.20.81.123:8282/js/three.js:6952
521.1 ms	2.66 %	521.1 ms	2.66 %	(program)	
52.7 ms	0.27 %	153.1 ms	0.78 %	▶ websocket.onmessage	172.20.81.123:8282/diswswebgl.html:128
9.1 ms	0.05 %	81.1 ms	0.41 %	▶ heartbeat	172.20.81.123:8282/js/heartbeat.js:3
77.0 ms	0.39 %	77.0 ms	0.39 %	(garbage collector)	
8.1 ms	0.04 %	63.9 ms	0.33 %	▶ munitionHeartbeat	172.20.81.123:8282/diswswebgl.html:841
7.1 ms	0.04 %	7.1 ms	0.04 %	set name	
1.0 ms	0.01 %	1.0 ms	0.01 %	checkExisting	172.20.81.123:8282/diswswebgl.html:680
1.0 ms	0.01 %	1.0 ms	0.01 %	localToGlobal	172.20.81.123:8282/diswswebgl.html:707

Figure 33. Tank game profile using WebSocket framework in Chrome.

Tree (Top Down)			
Self	Total	Function	
56.18 %	56.18 %	(idle)	
0.15 %	31.48 %	▼ renderFunc	frame?sourceid=...-static%2...:551
0.68 %	27.88 %	▼ Tank.update	172.20.81.123:8282/js/tank.js:81
4.43 %	14.78 %	▶ intersectObject	rs=AltRSTMVT8zd...VadqaSe5w:7021
0.46 %	12.33 %	▶ THREE.Raycaster.intersectObjects	rs=AltRSTMVT8zd...VadqaSe5w:7422
0.02 %	0.04 %	▶ THREE.Quaternion.multiplyQuaternions	rs=AltRSTMVT8zd...2VadqaSe5w:941
0.02 %	0.03 %	▶ THREE.Quaternion.multiply	rs=AltRSTMVT8zd...2VadqaSe5w:928
0.01 %	0.01 %	THREE.Quaternion.setFromAxisAngle	rs=AltRSTMVT8zd...2VadqaSe5w:793
0.01 %	0.01 %	THREE.Vector3.copy	rs=AltRSTMVT8zd...VadqaSe5w:1528
0.10 %	3.26 %	▶ render	rs=AltRSTMVT8zd...adqaSe5w:23504
0.04 %	0.11 %	▶ THREE.Object3D.add	rs=AltRSTMVT8zd...VadqaSe5w:7746
0.02 %	0.04 %	▶ update	172.20.81.123:8282/js/controls/OrbitControls.js:226
0.02 %	0.02 %	THREE.Quaternion.setFromAxisAngle	rs=AltRSTMVT8zd...2VadqaSe5w:793
0.01 %	0.01 %	renderPlugins	rs=AltRSTMVT8zd...adqaSe5w:23684
0.01 %	0.01 %	THREE.Vector3.set	rs=AltRSTMVT8zd...VadqaSe5w:1468
0.01 %	0.01 %	Munition.update	172.20.81.123:8282/js/munition.js:22
5.04 %	5.04 %	(program)	
0.07 %	2.48 %	▶ heartbeat	frame?sourceid=...-static%2...:907
0.03 %	2.13 %	▶ munitionHeartbeat	frame?sourceid=...-static%2...:946
0.02 %	1.38 %	▶ fr.onload	172.20.81.123:8282/js/peer.js:1323
0.02 %	0.75 %	▶ _dc.onmessage	172.20.81.123:8282/js/peer.js:1803
0.41 %	0.41 %	(garbage collector)	
0.08 %	0.11 %	▶ updateEntityMesh	frame?sourceid=...-static%2...:852
0.03 %	0.03 %	set name	
0.01 %	0.01 %	THREE.Object3D.add	rs=AltRSTMVT8zd...VadqaSe5w:7746

Figure 34. Tank game profile using WebRTC framework in Chrome.

C. NETWORK LOADING TIME

Besides the JavaScript performances, developer tools also provide function to record networking behaviors. The recording network information included file name, path, status, size, loading time, etc. When a browser visited a website initially, it downloaded cached web contents from the web server to the local disk. The purpose of caching web contents was to reduce network loading, because if the browser visited the same website in a specific period-of-time, the browser would verify the status of web contents to decide whether to download the web content via the network or access the content from the local disk. For example, if a file status showed 200, it meant the file was new to this browser and it was downloaded via the network. If a file status showed 304, it meant the file had not been modified since the last time it was cached, so the browser

would access the file from the local disk. These numbers are HTTP status codes that can be looked up on the Internet. In this thesis, the web server's default time of caching web content was 604,800 seconds, or one week. All the performance tests and tank games were run in a closed networking environment, so the speed of loading web contents was very fast. The total loads of the tank game were around 6 MB for a single player (see Appendix D). Most of the data consisted of graphical models and textures. This thesis used a simple tank model and rough terrain model to create a virtual world in browsers. Figure 35 and Figure 36 are screenshots of the tank model and the terrain models. The file sizes of these two models (hovertank10.js and bterrain.js) were not large; however, the textures for these two models were relatively larger than other necessary archives. The tank's texture was 1.1 MB, and the terrain's texture was 2.5 MB. Other JavaScript files (including three.js, peer.js, dis.js, etc.) only made up a small percentage of the total web contents.



Figure 35. Screenshot of tank model.

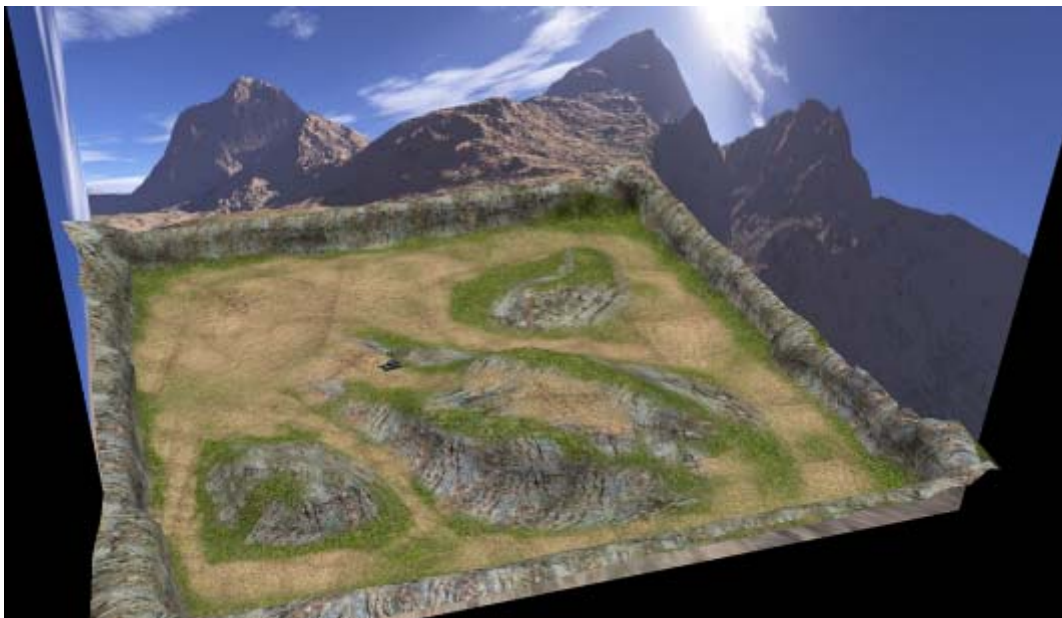


Figure 36. Screenshot of terrain model.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. RESULT DISCUSSIONS

According to the sending and receiving ability tests, this thesis found that different combinations of browsers and different networking frameworks had various capabilities of sending and receiving DIS PDUs. The outcomes showed that estimates of the PDU exchange rate should be based on the rate of PDU receipt. By analyzing the senders' memory change, it was discovered that sending pages buffered their sending data in their memory, instead of sending the data immediately. Although senders queued the sending data in their memories, Chrome and Firefox handled their sending events differently, which made for different efficiencies. Secondly, analyzing the profiling results showed that browsers spent their resources mostly on WebGL components that included rendering 3D models and computing intersections between 3D objects. The interchange of DIS messages did not account for a big usage of browser resources. Thirdly, the web contents that necessarily were downloaded from the web server to local browsers did not give any trouble in a local network in the experiments. Downloading web contents by remote users might cause latent problems such as slowing download or losing data if the necessary files were on the Internet or public network with poor connection, which might also affect the capability of exchanging DIS messages. Finally, according to the performance tests, the scale of web-based simulation using DIS protocol can be proposed.

A. SENDING AND RECEIVING EFFICIENCY

According to the performance tests, WebSocket framework had consistent receiving rates in both Chrome and Firefox, because the receiving rates were based on the outputting rate from the WebSocket gateway server. After taking a deeper look at WebSocket gateway server, the server's receiving rate of DIS messages was close to the receiving rates in recipient browsers, which indicated the WebSocket gateway server was efficient in transferring DIS packets. Additionally, this server neither use multicast nor broadcast, which were the

customary ways to distribute DIS packets on the UDP socket. Instead, the server delivered DIS PDUs to clients one by one because WebSocket is constructed on top of the TCP socket. For example, if there are only two clients connecting to the server, client A and client B, when client A is sending DIS messages, the server only distributes the receiving data to client B. If there are three or more clients, the server has to distribute the receiving data to all connected clients one-by-one, except for the sender itself. Therefore, the server distributing data to multiple clients decreased the receiving efficiency on recipient clients.

PeerJS had different performances in different browsers. Implementing the WebRTC, PeerJS used peer-to-peer architecture to transfer data, so that the sending rate would correspond to the receiving rate. The above results, however, indicated that actually the sending rates should refer to the receiving rates. The results also showed recipients had better receiving capabilities when using Firefox to send data. The method of distributing data in PeerJS was one-by-one—which was the same with the WebSocket gateway server—but the WebSocket framework used a central server to distribute data. The WebRTC framework is set up so that every participant can distribute data to their connected peers, and if there are multiple connections to a peer, the receiving rates on connected peers will decrease in an inverse ratio (i.e., peer A can send 2,000 PDUs per second to peer B in a one-on-one situation). If peer A has to send to two connected peers, peer B and peer C, the receiving rate in both peer B and peer C will be 1,000 PDUs per second.

By analyzing the receiving rates between browsers, this thesis found Firefox had better performances for sending data authentically in both frameworks. Figure 37 presents comparisons of receiving rates between different networks with different combinations of browsers. The x-axis represents the time for receiving 10,000 DIS PDUs, and the y-axis represents the first ten 10,000 DIS packets. WebSocket and WebRTC were the networking architectures for data transferring. When senders started to send 10,000 DIS PDUs for twenty times, receivers should have begun to receive data immediately. The figure shows,

however, that when data was sent from the Chrome browsers, the recipients took longer to receive the first 10,000 PDUs. Another expression of these data in linear form is in Appendix E.

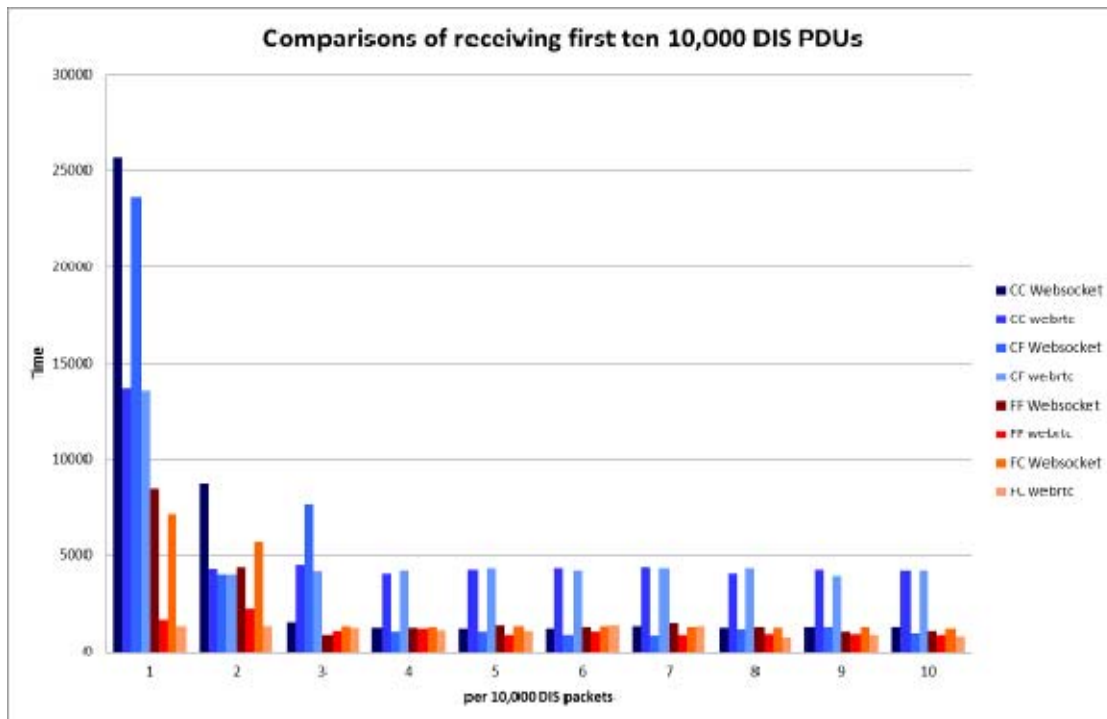


Figure 37. Comparisons of receiving first ten 10,000 DIS PDUs.

This thesis attempted to send a huge amount of ESPDUs (50,000 ESPDUs for 50 times), which crashed the Chrome sender after flushing and filling all the memory, and the receiver stopped receiving DIS packets. Firefox reacted in a different way. Although the Firefox browser indicated that it was not responding to the sender, the receiver continued to receive DIS PDUs persistently from a slow rate to a fast rate, allowing Firefox to successfully receive all 2,500,000 DIS packets after a long period-of-time. Furthermore, the transferring data was not lost in the WebSocket framework, and was only rarely lost in the WebRTC framework, which might be because the WebSocket is based on the TCP socket, while the WebRTC used the SCTP socket to configure data. If there was trouble with sending data in the WebSocket framework, then the

WebSocket would close, which meant that the webpage had to be reloaded or a WebSocket reconstructed in the browser. If the trouble happened in PeerJS, it would complain that the DataConnection object could not send data, but the data channel was still on; the browser could continue to send data after dropping all the unsent messages.

B. JAVASCRIPT PERFORMANCE

Profiling of the tank game showed that most of the browser resources were consumed on WebGL components. The sending and receiving functions only occupied small portions of the total percentage of browser resources. To see the detailed performances of each JavaScript function, Chrome and Firefox developer tools provide functions to see the explicit times of each function used. This capability could help application developers and designers know how much time is spent on each function.

1. Chrome Developer Tools

The profiling sample of Figure 33 showed the Chrome browser consumed 32.99% of the total time to run the renderFunc function, which spent 6455.8 ms during the entire period-of-time. This function, however, only spent 28.4 ms on itself. The rest of the time was spent on other functions that were invoked by this function. Chrome Developer Tools also provides Flame Chart view, which is a visual representation of JavaScript performance over time that thus offers a different way to view the profiling data. It also provides the running time of each function with single invoking, which is different from the aggregated time of running a specific function repeatedly. Figure 38 is a sample of invoking one renderFunc function in Flame Chart view. The information of the Flame Chart view included the name of this function, self time, total time, URL, aggregated self time, and aggregated total time. The aggregated total time of the renderFunc function was 6.46 s that corresponded to the time 6455.8 ms in Figure 33. Figure 39 was a portion of the recording profile that included the websocket.onmessage and the heartbeat functions. Looking at the details of this flame chart, found that

the running times of both functions, respectively, were less than 1 ms per invoking, and the aggregated total time of the heartbeat function was 81.103 ms, which was much shorter than 6.46 s.

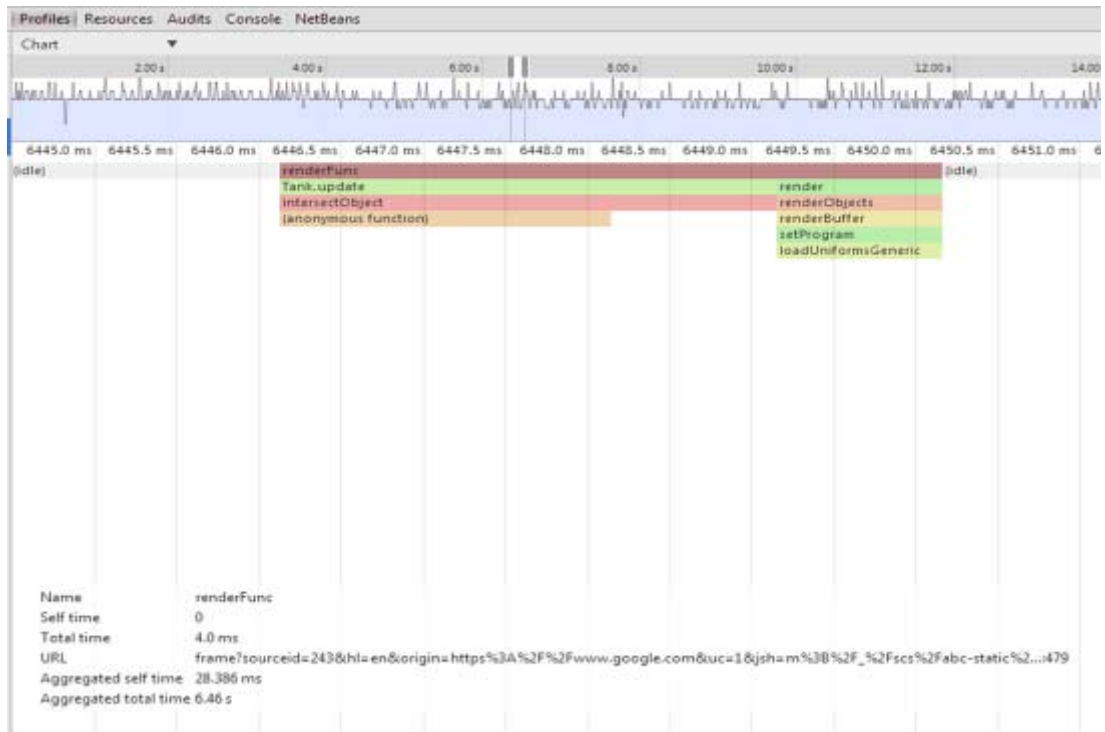


Figure 38. A Chrome profiling sample of invoking one renderFunc function.

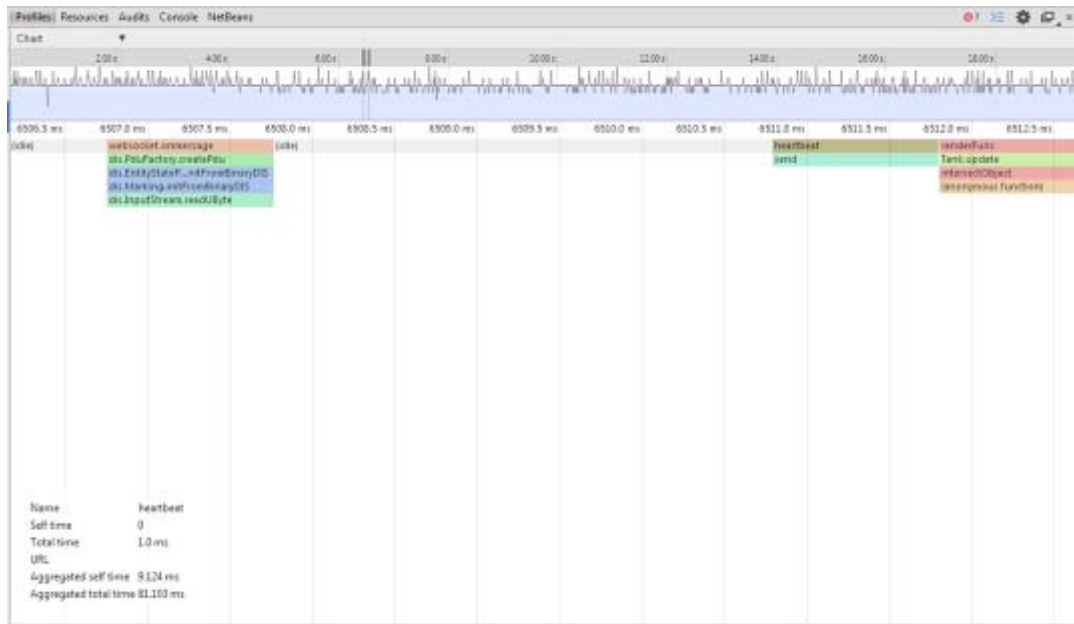


Figure 39. A Chrome profiling sample of invoking one websocket.onmessage and one heartbeat function.

2. Firefox Developer Tools

Firefox developer tools also have the ability to look up the explicit time of each function consumed. Figure 40 is a small sample of profiling the tank game with two players in the WebRTC environment. This sample range was from 2,353 to 2,403 of the complete profile, and the total running time is 49ms. If the running time is greater than the self, then there is an arrow at the left of the function's name to expand the function tree. In this example, the renderFunc() function spent 20 ms in running time, but the self time was zero, which means this function was expandable. This example also showed that five renderFunc() functions were executed in this 49 ms, which corresponded to the 10ms rendering rate. The executing time of each renderFunc() can be found by selecting one of these five functions and expanding this selected function to see the details of self time.

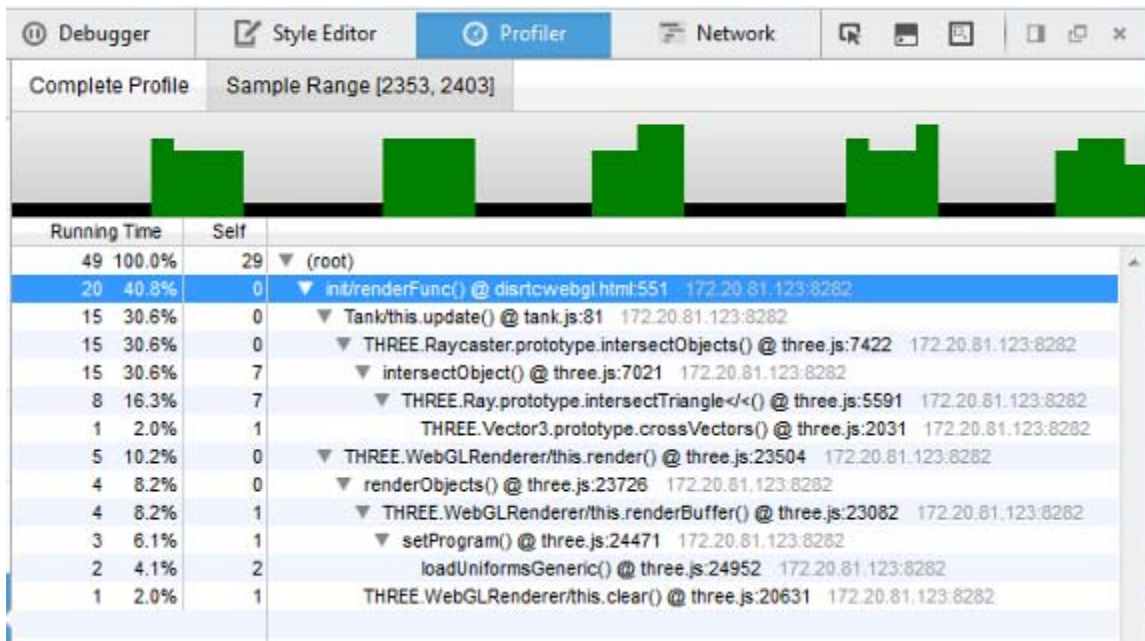


Figure 40. A Firefox profiling sample of invoking renderFunc functions.

After analyzing the JavaScript performances of the tank game, both profiling tools showed that even though the heartbeat and the munitionHeartbeat functions were invoked at the same rate with the rendering function, the browser devoted a large part of its resources to executing WebGL components rather than exchanging DIS PDUs. Therefore, the native capabilities of both WebSocket and WebRTC in browsers are competent enough to support web-based DIS simulations.

C. LOAD OF NECESSARY ARCHIVES

Loading page content from the server was no problem in this demonstration tank game because of the following reasons. First, all the tests were executed in a local networking environment. Second, the distributed files were only around 6M B for each client, and there were not many clients. Third, the client computers had very powerful CPUs and graphic cards. If someone wants to improve the sophistication of the tank game, however, then the archival size of models and textures must be increased. The sophistication of the model also increases the burden of computational power in clients. There are many

existing 3D models in various formats on the Internet, and it is easy to find a sophisticated model over 10 MB. Although web browser has capability to cache files from web server, it still has to load every necessary archive at the first time. If there are several large-size 3D models that have to be downloaded as web contents, the data distribution via public network would become a fundamental issue for building up web-based simulations. There are some popular techniques for reducing the impact of loading large data into web page such as minify JavaScript, apply Content Delivery Network, remove duplicate scripts (Souders, 2007).

D. GAME SCALABILITY

According to the results from performance tests, receiving rates were the major references for interchanging DIS PDUs in the network. Hence, the receiving rate can be used to scale the size of web-based multi-player games or simulations. For example, the lowest receiving rate in the experiments was around 1,900 PDUs per second, which was achieved when sending PDUs by Chrome browser in a peer-to-peer connection. Based on this result, a suggested number of players can be found by using equation (1).

Three variables that had to be considered: number of entity that sent ESPDU repeatedly, heartbeat rate, and number of clients; multiplying these three variables should result in a number less than 1,900. This tank game had two entities: tank entity and tank's ammunition entity; both entities sent ESPDU frequently. Each entity had its own heartbeat rate. Summing up the heartbeat rates multiplied by the corresponding entity would give the total amount of sending ESPDUs routinely, which excluded sending collision PDU and fire PDU. The game setting of the heartbeat frequencies for both tank and tank's ammunition was 10ms, which was equal to sending 100 ESPDUs per second. Therefore, the total amount of sending ESPDUs was around 200 per second. The last variable was the number of clients, because the sending technique was one-by-one on peer-to-peer connections and WebSocket gateway server. The

multiplier of the client number should be the total number of clients, minus the one that was the sending client itself. Back to the question, the suggested maximum number of players to play the tank game was 10 (the calculating result was 10.5 with the equation $(1*100+1*100)*(TC-1) = 1900$).

$$\left(\sum_{i=1}^n E_i * HR_i \right) (TC - 1) \leq 2000 \quad (1)$$

where,

n = total number of entities.

E_i = each entity

HR_i = heartbeat rate for the corresponding entity.

TC = total number of client

Ten players were not the maximum number playing the tank game. Different networking frameworks with different browsers had different capabilities of exchanging DIS packets. In addition, there are other ways to adjust the game capacity, such as changing the number of entities that frequently sent ESPDUs, and adjusting the heartbeat rate. One example was modifying the JavaScript program so that the tank's ammunition sent ESPDUs only when a tank fired, instead of the projectile sending ESPDUs periodically; the projectile would only send ESPDUs when its velocity was not zero. Another way was to adjust heartbeat rate on each entity. The rate of heartbeat affected game animation in participant's browsers because browsers used receiving ESPDUs to update entity positions and states. At 100 ESPDUs per second, the effective rendering rate is 100 fps in the remote participants, and it is not necessary to set an identical rate on both heartbeat and canvas rendering. In modern video games, 30 fps is qualified enough to be a commercial game, if this tank game lowered its heartbeat rates, the capacity of the number of players and/or the number of entities must be increased. Appendix F shows screenshots of multiplayer in the example tank game.

THIS PAGE INTENTIONALLY LEFT BLANK

VII. CONCLUSION AND FUTURE WORKS

A. CONCLUSION

The objectives of this thesis were to discuss and prove the feasibility of developing DIS web-based simulations using JavaScript. This thesis incorporated many web technologies and DIS protocol to outline infrastructures that supported web-based simulation. The major web technologies included JavaScript, WebSocket, WebRTC, and WebGL; there were many existing APIs that wrapped WebRTC and WebGL to help developers create web applications. This thesis suggested two different networking architectures, client-server and peer-to-peer, for interchanging DIS messages; and tested the capability of sending and receiving DIS PDUs between browsers. Furthermore, this thesis also integrated the above technologies to develop a browser-based tank game as a test application, and analyzed the performance of the tank game in different networking frameworks. At the same time, taking the advantage of cross-platform in browsers, both performance tests and the tank game can be executed and analyzed in Chrome and Firefox browsers, Internet Explorer and Safari did not support WebRTC yet. Other benefits of using web-based simulation included model reusing, collaboration, deployment, accessibility, and versioning control.

According to the performance tests and analysis of the demonstrating tank game, this thesis found that the modern web technologies are capable enough to construct web-based simulations. The performance tests included the capability of simply sending and receiving DIS PDUs in different networking frameworks; comparing the performance between the applying of WebGL materials; and cross-browser performances between Chrome and Firefox. The analysis of the tank game contained the resource consumption of rendering WebGL materials and interchanging DIS messages; and the page loads of necessary files from web server. The followings are the conclusions for the performance tests and the game analyses.

The performance tests showed that the speed of executing the sending ESPDUs function was faster than both WebSocket and WebRTC sent DIS messages. In fact, the measurements of sending rates were much higher than the receiving rates, meaning it was better to use receiving throughput as a reference to scale the size of web-based simulation. Both Chrome and Firefox have similar performances of sending and receiving DIS PDUs using the WebSocket framework. The performances of the WebRTC framework, however, favored Firefox. While using Firefox to send DIS packets via the WebRTC framework, both Chrome and Firefox had a better performance than any experiments with the WebSocket framework. While using Chrome to send the PDUs via the WebRTC, both Chrome and Firefox had worse performances than all the experiments with the WebSocket framework. Therefore, the Firefox and WebRTC combination is currently better than other combinations provided WebRTC is implemented using PeerJS.

This thesis used the worst-case performance of receiving rate (around 1,900 PDUs per second) as a guidance for scaling the tank game. The tank game had two entities that repeatedly sent 200 ESPDUs per second. The calculating result showed that this tank game was suitable for up to ten players. In addition, there were variables that could be adjusted to enlarge the capacity of the tank game, such as the heartbeat rate or the amount of entity that periodically sent ESPDU. The capacity of ten players was good enough to create an online game. For example, ten players can be divided into five-versus-five players. Famous examples of five-versus-five games include League of Legends and Defense of the Ancients (DotA) in Warcraft III; however, they are not web-based games.

This thesis used Chrome and Firefox developer tools to profile the tank game and recode the time of loading web contents. Profiling tools contained percentages of time that each function used, and explicit times of each function spent. The profiling results showed most of the browser resources were spent on WebGL components, and the function for exchanging DIS packets only used a

small portion of the browser resources (see Figure 33, Figure 34, Appendix C-1, and Appendix C-2). The times of loading web contents showed browsers received all necessary web contents quickly in a closed network (see Appendix D).

Based on the above results, the DIS protocol could be applied in web-based applications. Both WebSocket and WebRTC frameworks were capable of supporting a ten-player first-person-shooter game in a browser, and the size of the game was expandable by adjusting the variables. WebGL created 3D graphics shows in web browsers without any plugins, and the performance of WebGL was dependent on the power of computers and the degree of model complexity, which was a trade-off among budget, performance, and the quality of 3D models. Methods of using the DIS protocol and the game styles are varied. This thesis showed that web-based DIS simulation is workable, and simulation designers could refer to the basic capability of exchanging DIS messages to develop web-based simulation for different purposes.

B. FUTURE WORKS

This thesis sought quick ways of constructing networking environments to interchange DIS between different browsers, so simulation developers could focus on the functions and scenarios for different purposes. This thesis also focused on developing an example game as a practical application to verify the feasibility of web-based simulation. There were some experiments and improvements that were not performed in this thesis, such as: performance testing of mobile devices; improvement of the WebRTC sending capability in Chrome browsers; benchmark testing of the WebSocket and the WebRTC in different browsers; discussions of disadvantages of web-based simulations; and comparisons between web-based DIS simulations and traditional DIS simulations.

1. Performance Tests and Web Applications on Mobile Devices

The development of mobile devices has flourished in the past few years, and most of the mobile devices have installed web browsers such as iOS Safari,

Opera Mini, Android Browser, and Chrome for Android. One of the advantages of web applications is cross-platform, which indicates mobile devices can execute web applications in their compatible web browsers. Currently, almost every browser supports WebSocket, except Opera Mini; however, only Chrome for Android supports the WebRTC. This thesis did not measure the performance of interchanging DIS PDUs between browsers in mobile devices. This result would be a consideration for developing web applications, because the computational performance on mobile devices is typically slower than personal computers or laptop computers. Other considerations include WebGL components and input mechanisms. WebGL is one of the major elements that uses a lot of browser resources. So far only Chrome for Android supports WebGL, but the next version of iOS Safari will begin supporting WebGL(Deveria, n.d.-a). In addition, mobile devices usually do not attach to a keyboard or mouse, and the main input device for mobile device is a touch screen. Therefore, development of web applications for mobile devices has to take into consideration input mechanisms using graphical user interfaces (GUI) for users.

2. Improvement of WebRTC Sending Capability in Chrome Browser and Benchmark Test

According to the results in Chapter V, Performance Tests, the lowest receiving rates were from Chrome sent to Chrome and Chrome sent to Firefox. This thesis used PeerJS that wrapped WebRTC to create data channels between browsers. PeerJS is an easy and convenient API for developers to create web applications that apply WebRTC functions in a browser. However, because PeerJS is a wrapped API, it is difficult to modify or change the JavaScript code inside the API. Additionally, there is no benchmark application for testing WebRTC performance between Chrome and Firefox. Comparing with the receiving capability that sending from Firefox browser (Figure 29), it seems like Chrome should have space for improvement in the performance of the WebRTC framework. Therefore, further studies directed at WebRTC technology are required to improve the sending performance of the Chrome browser.

3. Disadvantages of Web-based Simulation

This thesis discussed the advantages of web applications, along with the establishment of infrastructures that exchange DIS messages, and the development of the tank game. Some disadvantages of web-based simulations were not studied or discussed in this thesis, such as security vulnerability, GUI limitation, and latency. In addition, further study and discussion should contain comparisons between web-based DIS simulation and traditional DIS simulation.

4. Feasibility of Other Web-based DIS Applications

This thesis only developed a first-person-shooter tank game as an example of a DIS simulation in a browser. However, applications that use DIS protocols are varied. Further research is needed to discover which web-based domains can be applied to DIS simulations. For example, the WebSocket gateway server can receive native DIS from a UDP socket, so browsers can receive DIS packets from other simulation systems such as VBS2. Is it possible to create a battlefield viewer by receiving DIS messages in browsers, or to create a simplified interactive game interface in browsers? On the other hand, WebRTC emphasizes real-time communication between browsers, and is designed for audio and video communication. Is it possible to incorporate audio and video in web-based DIS simulations? What is the benefit of incorporating audio and video in web-based simulations, and does it affect the performance of sending and receiving DIS PDUs?

5. Performance Tests in Public Networking Environments

All the experiments and performance tests in this thesis were done in a closed-networking environment. Both WebSocket and WebRTC were competent enough to support web-based DIS simulations; however, a public networking environment is much more complex. Both WebSocket gateway servers and WebRTC peers used one-by-one mechanisms to distribute DIS packets, but the bottom layers were different. The WebSocket was based on a TCP socket, and the WebRTC used a UDP socket. This thesis did not examine performances on a

public network or Internet environments. Additional experiments should be done to compare the performance between the WebSocket and the WebRTC on a public network.

APPENDIX A. TABLE OF ENTITY STATE PDU FIELDS

Appendix A shows the fields of entity state PDU, and DIS simulation uses ESPDU to communicate information about an entity's state.

Field size (bits)	Entity State PDU fields	
96	PDU Header	Protocol Version—8-bit enumeration
		Exercise ID—8-bit unsigned integer
		PDU Type—8-bit enumeration = 1
		Protocol Family—8-bit enumeration = 1
		Timestamp—32-bit unsigned integer
		Length—16-bit unsigned integer
		PDU Status—8-bit record
		Padding—8 bits unused
48	Entity ID	Site Number—16-bit unsigned integer
		Application Number—16-bit unsigned integer
		Entity Number—16-bit unsigned integer
8	Force ID	8-bit enumeration
8	Number of Variable Parameter Records (<i>N</i>)	8-bit unsigned integer
64	Entity Type	Entity Kind—8-bit enumeration
		Domain—8-bit enumeration
		Country—16-bit enumeration
		Category—8-bit enumeration
		Subcategory—8-bit enumeration
		Specific—8-bit enumeration
		Extra—8-bit enumeration
64	Alternate Entity Type	Entity Kind—8-bit enumeration
		Domain—8-bit enumeration
		Country—16-bit enumeration
		Category—8-bit enumeration
		Subcategory—8-bit enumeration
		Specific—8-bit enumeration
		Extra—8-bit enumeration
96	Entity Linear Velocity	<i>x</i> -component—32-bit floating point
		<i>y</i> -component—32-bit floating point
		<i>z</i> -component—32-bit floating point

192	Entity Location	X-component—64-bit floating point
		Y-component—64-bit floating point
		Z-component—64-bit floating point
96	Entity Orientation	Psi (ψ)—32-bit floating point
		Theta (θ)—32-bit floating point
		Phi (ϕ)—32-bit floating point
32	Entity Appearance	32-bit record
320	Dead Reckoning Parameters	Dead Reckoning Algorithm—8-bit enumeration
		Other Parameters—120 bits
		Entity Linear Acceleration— 3×32 -bit floating point
		Entity Angular Velocity— 3×32 -bit floating point
96	Entity Marking	Character Set—8-bit enumeration
		11, 8-bit unsigned integers
32	Capabilities	32-bit record
128	Variable Parameter record #1	Record Type—8-bit enumeration
		Record-Specific fields—120 bits
<ul style="list-style-type: none">•••		
128	Variable Parameter record #N	Record Type—8-bit enumeration
		Record-Specific fields—120 bits
Total Entity State PDU size = $1152 + 128N$ bits		
where		
N is the number of Variable Parameter records		

Table 6. Fields of entity state PDU.

A. APPENDIX B-1. FIREFOX PROFILING OF SENDING ESPDUS

[illegible]

Figure 41. Firefox profiling of sending ESPDU in the WebSocket framework.

B. APPENDIX B-2. SENDING SECTION IN APPENDIX B-1

Appendix B-2 shows the sending section in Appendix B1.



Figure 42. Sending section in Appendix B1.

C. APPENDIX B-3. FIREFOX PROFILING OF RECEIVING ESPDUS

Appendix B-3 shows a result of using Firefox developer tools to profile the receiving behavior in the WebSocket framework for a short period-of-time.

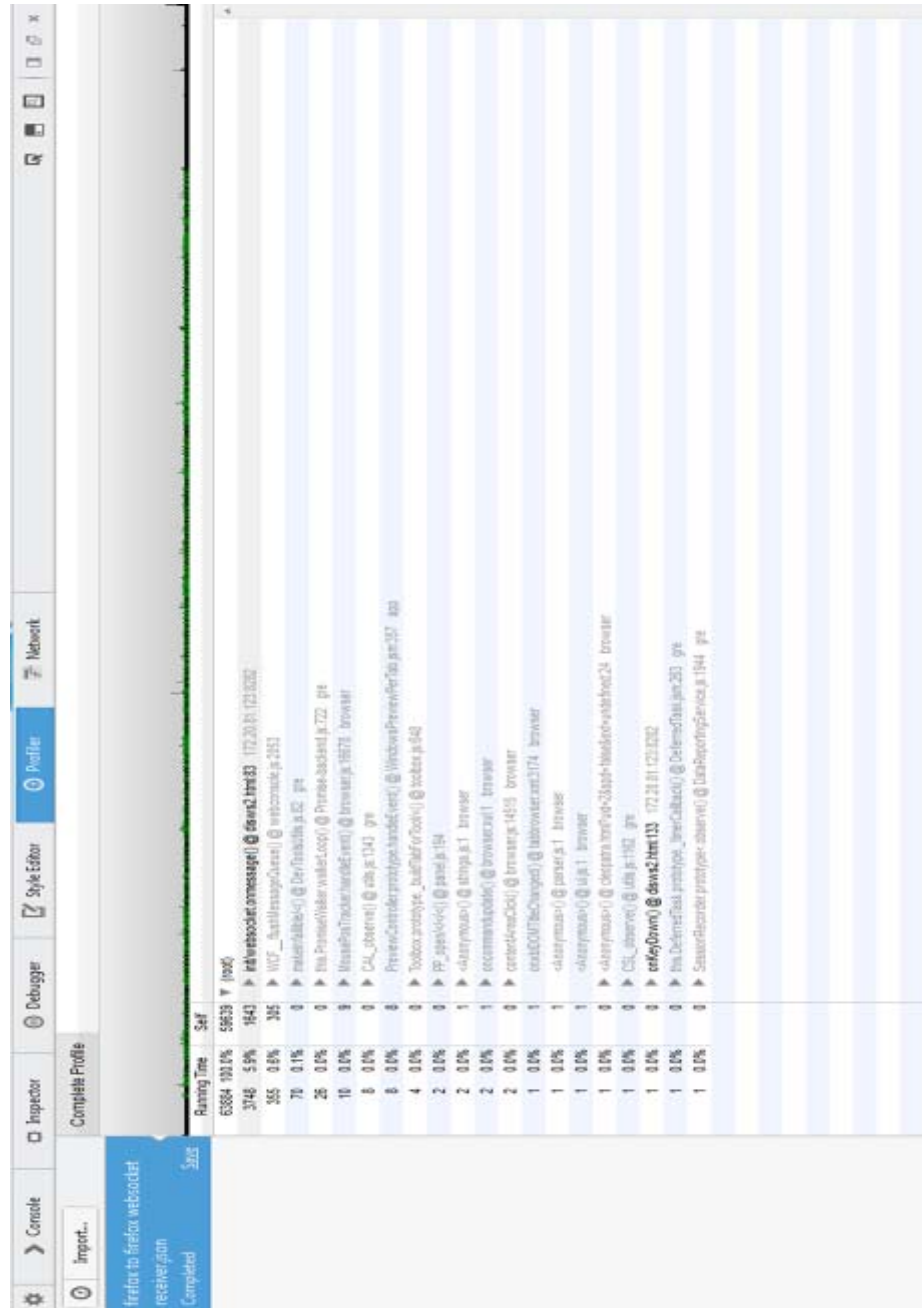


Figure 43. Firefox profiling of receiving ESPDU in the WebSocket framework.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C. TANK GAME RESULTS

A. APPENDIX C-1. FIREFOX PROFILING IN THE WEBSOCKET FRAMEWORK

Appendix C-1 shows a result of profiling the tank game using Firefox developer tools in the WebSocket framework.

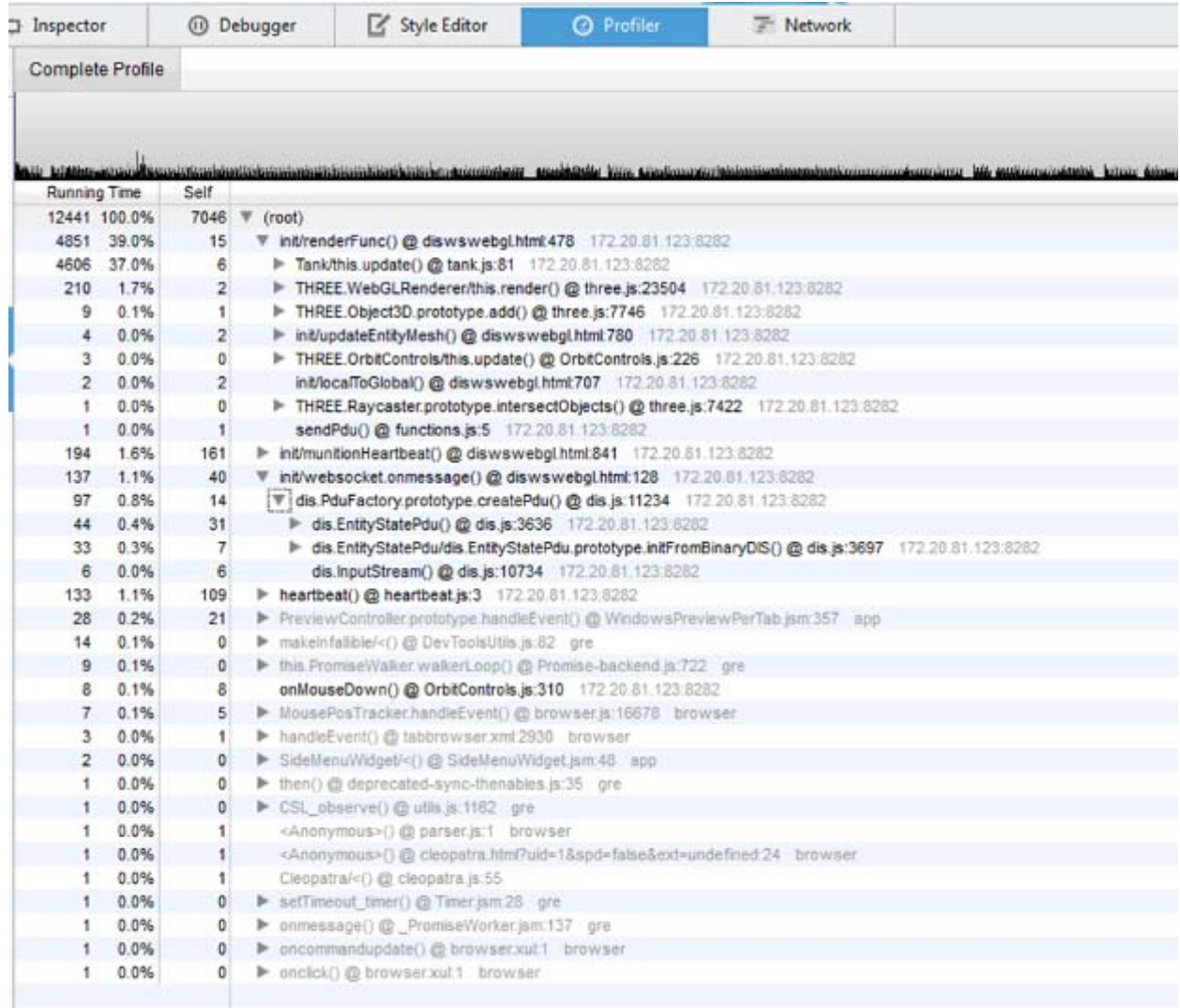


Figure 44. Firefox profiling of the tank game in the WebSocket framework.

B. APPENDIX C-2. FIREFOX PROFILING IN THE WEBRTC FRAMEWORK

Appendix C-2 shows a result of profiling the tank game using Firefox developer tools in the WebRTC framework.

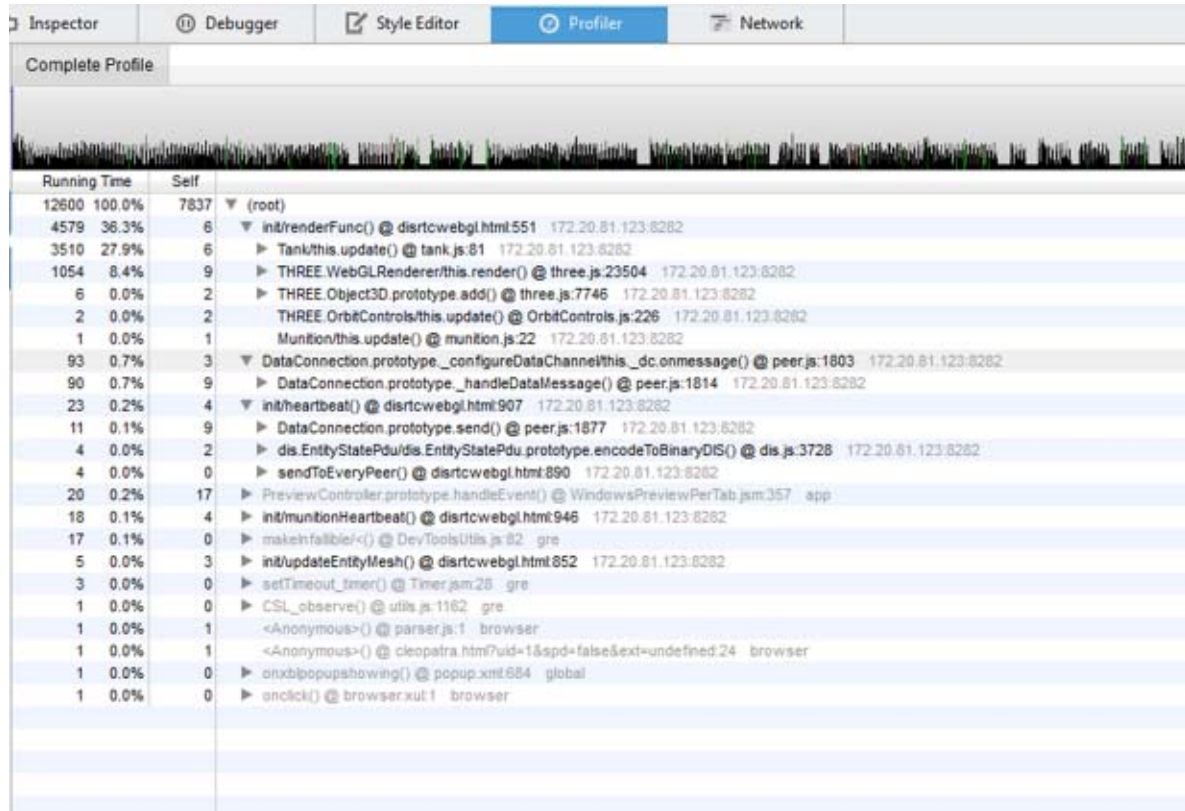


Figure 45. Firefox profiling of the tank game in the WebRTC framework.

APPENDIX D. NETWORK LOADING TIME

Appendix D is an example of recording the time of loading web contents.

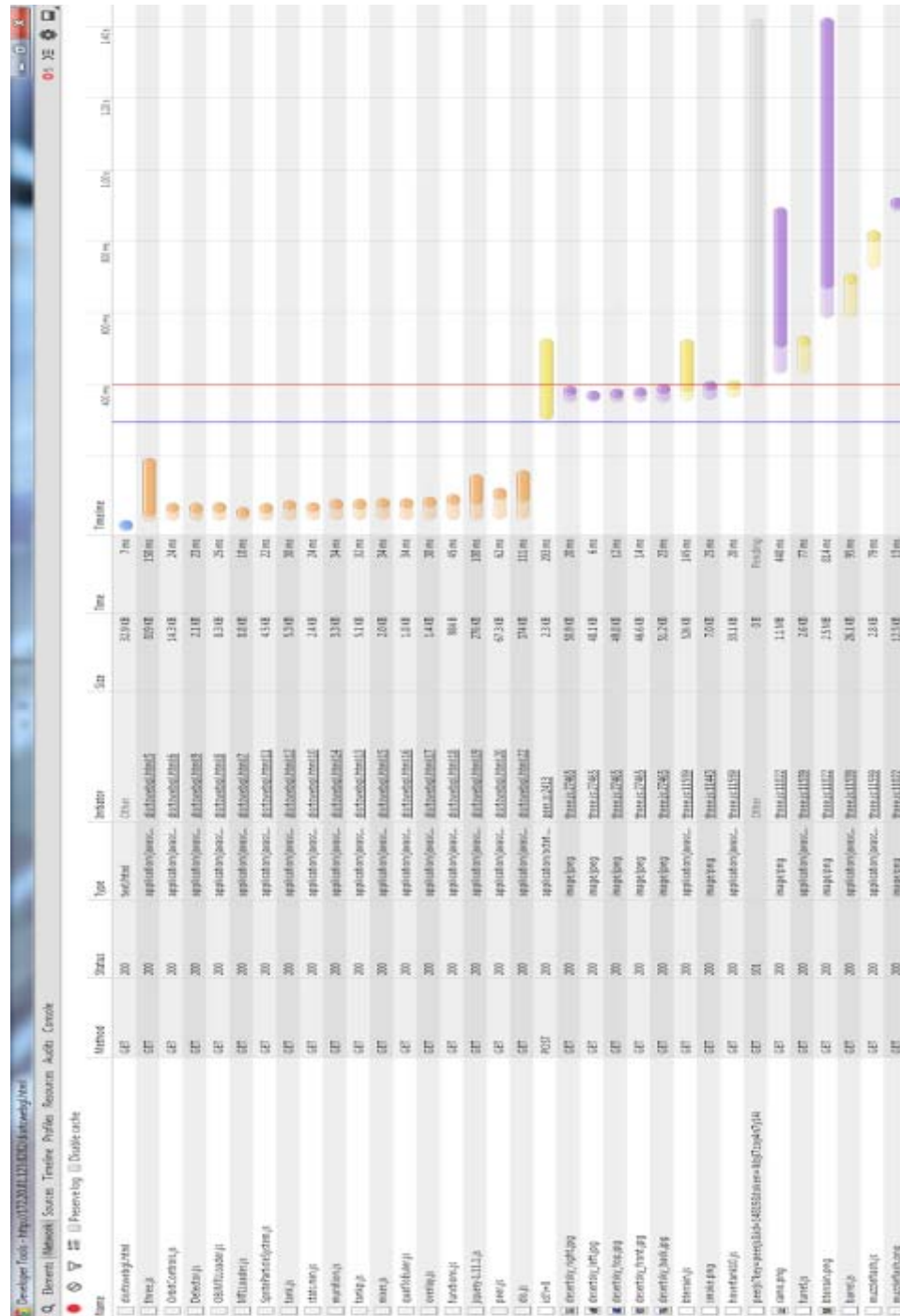


Figure 46. Network loading time.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX E. COMPARISONS OF RECEIVING FIRST TEN 10,000 DIS PDUS IN THE LINEAR FORM

Appendix E is the comparisons of receiving first ten 10,000 DIS PDUs in the linear form. It shows that the times of receiving first one 10,000 DIS PDUs were slower when the senders were Chrome.

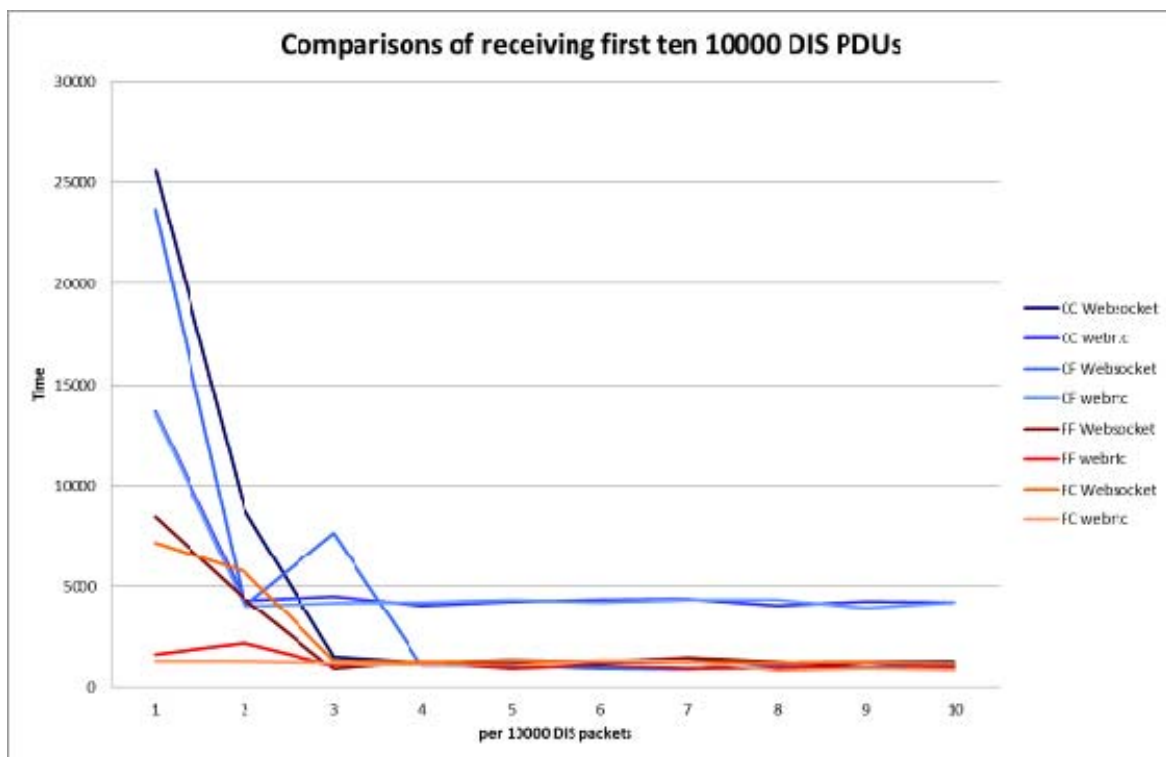


Figure 47. The comparisons of receiving first ten 10,000 DIS PDUs in the linear form.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX F. SCREENSHOTS OF MULTIPLAYERS IN THE TANK GAME

Appendix F shows that it is possible to create a web-based DIS simulation for many users.



Figure 48. Screenshots of multiplayer in the tank game I.



Figure 49. Screenshots of multiplayer in the tank game II.



Figure 50. Screenshots of multiplayer in the tank game III.



Figure 51. Screenshots of multiplayer in the tank game IV.

LIST OF REFERENCES

- Arewefastyet. (n.d.). Retrieved August 20, 2014, from <http://arewefastyet.com/>
- Axes conventions. (n.d.) *Wikipedia*. Retrieved August 29, 2014, from http://en.wikipedia.org/wiki/Axes_conventions
- Bu, M., & Zhang, E. (n.d.). PeerJS. Retrieved July 31, 2014, from <http://peerjs.com/docs/#api>
- Chrome V8. (n.d.). Retrieved August 20, 2014, from <https://developers.google.com/v8/intro>
- Deveria, A. (n.d.-a). Can I use WebGL. Retrieved August 4, 2014, from <http://caniuse.com/webgl>
- Deveria, A. (n.d.-b). Can I use WebRTC. Retrieved July 31, 2014, from <http://caniuse.com/#search=webrtc>
- Deveria, A. (n.d.-c). Can I use WebSocket. Retrieved July 31, 2014, from <http://caniuse.com/#search=websocket>
- Dutton, S. (2013). WebRTC in the real world: STUN, TURN and signaling. Retrieved from <http://www.html5rocks.com/en/tutorials/webrtc/infrastructure/>
- Farlane, D. M., Ryan, P., Davis, R., Rekkas, G., Davies, M., Ross, P., ... Carpenter, R. (2004). *Distributed simulation guide*. Canberra, Australia: Australian Defence Simulation Office.
- Firefox SpiderMonkey. (n.d.). Retrieved August 20, 2014, from <https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey>
- Fishwick, P. (1996). Web-based simulation: some personal observations. ... *of the 28th Conference on Winter Simulation* (pp. 772–779). Coronado, CA: Winter Simulation Conference.
- Flanagan, D. (2011). *JavaScript - The definitive guide*. (M. Loukides, Ed.) (Sixth., p. 1078). Sebastopol, CA: O'Reilly Media.
- Frame rate. (n.d.) *Wikipedia*. Retrieved August 2, 2014, from http://en.wikipedia.org/wiki/Frame_rate
- Grigorik, I. (2013). High performance browser networking. Retrieved from <http://chimera.labs.oreilly.com/books/1230000000545>

- JSON. (n.d.). Retrieved July 31, 2014, from <http://json.org/>
- Lautenschlager, S. (n.d.). Javascript Key Codes. Retrieved August 1, 2014, from <http://www.cambiaresearch.com/articles/15/javascript-char-codes-key-codes>
- McCall, J. M. (2012). *DIS IEEE Std 1278.1-2012 - Standard for Distributed Interactive Simulation –Application Protocols* (p. 729). IEEE.
- McGregor, D., Blais, C., & Brutzman, D. (n.d.). *A Javascript implementation of the binary DIS protocol*. Monterey, CA: MOVES Institute.
- McGregor, D., & Brutzman, D. (n.d.). *Networked virtual environments with Javascript, WebSockets and WebGL*. Monterey, CA: MOVES Institute.
- McGregor, D., Brutzman, D., & Grant, J. (2008). Open-DIS: An open source implementation of the DIS protocol for C++ and Java. *Simulator Interoperability Working Group (SISO) Fall* Retrieved from <http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:Open-DIS:+An+Open+Source+Implementation+of+the+DIS+Protocol+for+C++++and+Java#0>
- McGregor, D., Grant, J., Smith, P., Harder, R., & Snyder, S. (n.d.). Open-DIS. Retrieved from <http://open-dis.sourceforge.net/Open-DIS.html>
- Parisi, T. (2012). *WebGL: Up and running* (p. 230). Sebastopol, CA: O'Reilly Media.
- Powers, S. (2010). *JavaScript cookbook*. (S. St.Laurent, Ed.) (p. 528). Sebastopol, CA: O'Reilly Media.
- Profiling JavaScript Performance. (n.d.). Retrieved July 31, 2014, from <https://developer.chrome.com/devtools/docs/cpu-profiling>
- Ristic, D. (2014). WebRTC data channels. Retrieved from <http://www.html5rocks.com/en/tutorials/webrtc/datachannels/>
- Rogerson, S. (1997). Implementation of a distributed interactive simulation interface in a Sea King Flight Simulator. Retrieved from <https://tspace.library.utoronto.ca/handle/1807/11813>
- Simulation Interoperability Standards Organization (SISO) Reference for : Enumerations for Simulation Interoperability*. (2013).
- Souders, S. (2007). *High performance web sites* (p. 146). Sebastopol, CA: O'Reilly Media.

Steed, A., & Oliceira, M. F. (2010). *Networked graphics—Building networked games and virtual environments* (p. 522). Burlington, MA: Morgan Kaufmann.

w3schools. (n.d.). JavaScript tutorial. Retrieved July 31, 2014, from <http://www.w3schools.com/js/>

WebRTC. (n.d.). Retrieved July 31, 2014, from <http://www.webrtc.org/>

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
Ft. Belvoir, Virginia
2. Dudley Knox Library
Naval Postgraduate School
Monterey, California